27/28

NOTE DI SOFTWARE

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

Honeywell
Honeywell Information Systems Italia

Note di software.
è un bollettino trimestrale, edito a cura del Centro di Ricerca e Progettazione della Honeywell Information Systems Italia e dell'Istituto di Cibernetica dell'Università di Milano.

Note di software.
intende costituire uno strumento per la rapida e informale circolazione di informazioni, idee ed esperienze nell'area della tecnologia del software. In particolare, intende stimolare il dibattito sulle metodologie orientate alla diminuzione del rapporto costo/qualità dei programmi, e sugli strumenti atti ad automatizzare l'attività di produzione del software in ogni suo aspetto.

La collaborazione a **Note di software** è libera.
In particolare, si sollecitano contributi nella forma di brevi monografie e bibliografie essenziali commentate sui seguenti temi: tecniche e strumenti per la programmazione strutturata e modulare, metodologie e strumenti per il testing e il debugging dei programmi, documentazione del software, aspetti organizzativi e gestionali nella costruzione di sistemi software di grandi dimensioni.

Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Comitato di Redazione (HISI - Via ai Laboratori Olivetti - Pregnana Milanese), o di inviare direttamente il dattiloscritto in triplice copia.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione vengono revisionati dal Comitato di Redazione, che si avvale del contributo di specialisti del settore. Ogni contributo pubblicato riflette, comunque, le opinioni dei suoi autori, e non necessariamente quelle del Comitato di Redazione. Eventuali revisioni saranno effettuate di comune accordo fra il Comitato di Redazione e i proponenti.

Note di software.
Viene distribuito ad aziende, Enti e specialisti del settore che ne facciano richiesta alla redazione.

Direttore responsabile: Paolo Ghelfi
Comitato di Redazione: Roberto Polillo - Wanda Anselmetti - Stefania Bandini
Redazione: Franco Soresini

ARCHIVIO STORICO
ASSOCIAZIONE POZZO DI MIELE
FPDM-81 RNSW118

27/28

FPDM-81

NOTE DI SOFTWARE

Giugno 1985

Indice:

QUESTO NUMERO... Pag. 1

- QUALITATIVE MODELS OF PHYSICAL SYSTEMS, di L. Spampinato » 3
- COMPUTAZIONE PARALLELA: UNA PANORAMICA, di F. De Santis, A. Negro e E. Pisano » 15
- H - UN LINGUAGGIO DI PROGRAMMAZIONE LOGICA, di R. Ghislanzoni, V. Samek Ludovici e G. Tornielli » 23
- UN LABORATORIO PER LA COMPRESSIONE DI TESTI CON TECNICA WORD MAPPING, di O. Bosatelli e M. Bosetti
- UN SISTEMA PER LA DOCUMENTAZIONE DI PROGRAMMI DIDATTICI, di M. Ferrari e H. Le Van » 43

Autorizzazione del Tribunale di Milano n. 87 del 23 febbraio 1977

IN QUESTO NUMERO...

Due articoli di rassegna aprono questo numero doppio di NOTE DI SOFTWARE. Si tratta di due lavori che toccano argomenti di grande attualità non solo riguardo alla ricerca, ma anche rispetto a settori applicativi d'avanguardia.

Nel primo, infatti, L. Spampinato presenta una rassegna dei risultati nel campo della modellazione di sistemi fisici basata su criteri qualitativi. È questo un tema di grande interesse sia negli studi sull'Intelligenza Artificiale che nella realizzazione di Sistemi Esperti e nelle applicazioni della robotica.

F. De Santis, A. Negro e E. Pisano, nel secondo articolo di rassegna, presentano alcuni degli aspetti fondamentali che riguardano le computazioni parallele: oltre ad una introduzione al problema della parallelizzazione dei processi computazionali, gli autori pongono la loro attenzione sul disegno degli algoritmi paralleli.

Questo numero di NOTE DI SOFTWARE prosegue con altri tre lavori più specifici.
L'articolo di R. Ghislanzoni, V. Samek Ludovici e G. Tornielli è dedicato ad un linguaggio di programmazione logica. Si tratta di H, un linguaggio basato sulla logica del primo ordine. Il lavoro introduce alcuni concetti fondamentali che caratterizzano le famiglie dei linguaggi logici e, in appendice, fornisce il codice dell'interprete di H.

O. Bosatelli e M. Bosetti presentano poi un laboratorio per la compressione di testi in ambiente UNIX. Dopo una parte introduttiva sulle tecniche di compressione corrente, gli autori descrivono una loro implementazione che utilizza tecniche di word-mapping.

Infine, M. Ferrari e H. Le Van presentano MIDA, un sistema di documentazione con capacità di browsing nato nell'ambito del Laboratorio di Scienze dell'Informazione dell'Università di Milano. MIDA comprende una metologia per la documentazione di programmi da parte degli studenti.

La Redazione

QUALITATIVE MODELS OF PHYSICAL SYSTEMS

Luca Spampinato
Quinary Milano - Istituto di Cibernetica - Università di Milano

ABSTRACT

The following pages contain an overview of some relavant research results in the field of qualitative modeling of physical systems. This survey covers the issues related to the modeling of non digital system and in particular those of them more closely related with studies about commonsense reasoning on physical systems and complex mechanical devices. [10].

RIASSUNTO

È questa una rassegna di alcuni dei risultati più rilevanti nel campo della modellazione di sistemi fisici basata su criteri qualitativi. È questo un campo oggetto di crescente interesse nell'ambito degli studi sull'intelligenza artificiale.
L'opinione di molti è infatti che questa sia la via giusta per affrontare l'analisi del cosiddetto "ragionamento a senso comune". In particolare vengono trattati gli aspetti legati alla modellazione e alla rappresentazione di conoscenza relativa ai sistemi non digitali.

1. MOTIVATIONS

Qualitative modeling is concerned with the description of physical situations and their changes. Qualitative descriptions are useful for both humans and mechanical intelligent systems. A qualitative description enables an intelligent system to perform a kind of commonsense reasoning about the behaviour of complex physical systems. As commonsense knowledge about the physical world is often incomplete, the qualitative models must be able to work even if they embody incomplete knowledge and it must be possible to enhance them in a incremental way. In 1979, P.J. Hayes expressed in his "Naive Physics Manifesto" [1], a great number of ideas - already shared by some artificial intelligence people - about the opportunity to build a theory of qualitative modeling of physical systems.

The "Manifesto" contains a great amount of compelling reasons and very general ideas and we refer to it for a complete discussion on the motivations for a naive physics. Here we are primarily concerned with the basic ideas behind such a theory.

Systems that must deal with the physical world, in terms of both reasoning and actions, have to embody a great amount of knowledge about it. Artificial intelligence has been rather unsuccessful with physics problem solving. In particular, the KBS field has produced a number of systems that can solve tasks that require a deep human expertise. However, such systems are often constrained by the narrow range of their expertise. The representation of knowledge embodied in the system is often influenced by problem recognition and by classification criteria. Knowledge is fully available for problem solving when the system has to face situations which can be classified as standard, in a sense. The current expert systems have no commonsense and aren't able to recognize all the situations in which their knowledge could be useful. General knowledge about the application domain, though available, is not fully exploited, thus unusual, unexpected problems, though very simple, make the system useless.

An intelligent system concerned with physical problems should be able to perform a number of typical tasks in the following areas:

Simulation
Starting from a structural description of the physical system and from some initial conditions, it must be able to determine a likely course of future behaviour of the system itself. It should also be able to manage possible ambiguities due to qualitativeness and possible incompleteness of the description. In fact, an incomplete description does not guarantee that it defines a unique physical system and a unique behaviour

for it. This concept will be clearer when some technical details are provided about the meaning of the term "qualitative".

Envisionment
Starting with a structural description the system can determine all the possible behavioural sequences. This process has to meet two important requirements:

completeness
All the possible envisioned behaviours can be realized in the real system with some choice of the parameters.

realizability
All the possible real behaviours are envisioned.
In a sense, envisionment can be seen a "total simulation" which, with the above constrains, recognized all the admissible states for the physical system.

Explanation
A compelling and casual explanation of the physical behaviour must be provided in terms of changes and information/constrains propagation. In other words, an explanation must be concerned with the way the physical system achieves its behaviour and must be expressed in terms of influences among changes in the parameters values.

Diagnosis
A fault in a physical system can be observed only when the system behaviour is different from a specified, desired or natural one. A typical diagnostic task is to recognize the change in the structure which determine the difference in behaviour.

Functionally deduction
The function of a physical system in the relation between its behaviour and the goal of a human user. Starting from a structural and behavioural description of the physical system, the function of the components must be deduced in terms that are common in the application field. Commonsense reasoning is often based on related functions rather than on related behaviours. In other words, while the description of the system is inherently structural and behavioural, information must be provided to the user in terms of functionality of each component.

In order to give expert systems a kind of commonsense problem solving capability, traditional quantitative physics does not seems suitable for a number of reasons:

- Quantitative models of modern physics are related to very abstract concepts (functional spaces, Lagrange of Hamilton functions, etc.), very far from the common intuition of the everyday world. The expressive power of such theories relies on a great number of unstated conventions that could be defined "pre-physical" and which are mathematical and philosophical. Moreover their interpretation subsumes a lot of implicit commonsense knowledge.

- In quantitative modeling, parameters and derivates are time-varying, real-valued quantities; thus reasoning about complex systems requires a great amount of numerical computation.

- Quantitative descriptions are useful only if complete knowledge about the parameters behaviour is available in terms of numerical values. Reasoning about physical systems, human experts often lack precise numerical values for many parameters and some parameters can be difficult to measure. Predictions must often be based on qualitative indications about the parameters trends.

- A quantitative analysis of a physical system can be performed only provided that the system of differential equations which describes it has closed solutions.
  Roughly speaking, this means that a quantitative description must be complete.
  An application-oriented physical problem solver, on the contrary, has to be able to deal with incomplete descriptions.

It is clear that humans are very skilled in dealing with problems in the physical world, but we have no theory of such human capabilities which can be used in transferring them to computers. A deep investigation in qualitative modeling as a theory is hence very important.

2. COMMON BASES

The main goal of naive physics is to establish a new framework to represent knowledge about physical systems, which could be more useful for commonsense reasoning purposes than systems of differential equations. In this framework, a central role is played by a kind of formal reasoning about the behaviour of physical systems. The naive physics approach is aimed at reducing the quantitative precision of the behavioural description still retaining the crucial distinctions. In other words, behavioural descriptions are centered around changes in physical situations and thus only information related with changes must be recognized in quantitative terms.
Precise parameters values are considered only in correspondance of the significant changes in the physical situation.

Traditional quantitative approaches generally start from an analysis of the real system and lead to build a model constitued of differential equations and of some boundary conditins. This is the physicist's task. The equations must then be solved, by mathemati-

cians, with either analytical or numerical methods, and the solution, stated in terms of real functions, must be interpreted in order to give an idea of the behaviour of the system.

Qualitative approaches lead, on the contrary, to build a structural description, taking knowkedge both from an analysis of the system and from the quantitative model wich is often well-known. A description of the behaviour of the system can then be computed by means of a qualitative simulation. By analogy with the traditional approach, this activity can be defined qualitative mathematics. Such simulation is performed by means of a set of formal and semi-formal methods and tools which allow managing the qualitative nature of involved information.

Qualitative simulation is a symbolic manipulation process that can be considered an inference activity. In particular an interesting part of the inferential framework must be devoted to a quantitative form of arithmetic, a symbolic calculus very similar to the ordinary arithmetics but specially suited for manipulating qualitative and approximative quantities. The above concepts are shown in Figure 1.

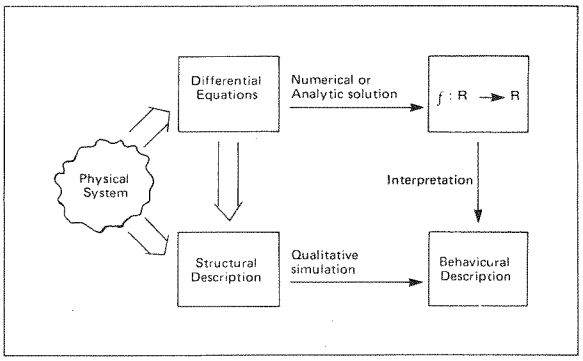

Fig. 1 - The qualitative structural description is capable of capturing a less complete state of knowledge than a differential equation, and the qualitative simulation produces a partial description of the mechanism's behaviour. Because the qualitative simulation uses heuristics, the two paths through the above diagram do not always yield the same result.

Current research in qualitative modeling addresses both qualitative physics and qualitative mathematics. Qualitative simulation involves the use of some tools based on heuristics; these often rely on a number of intuitive commonsense assumptions which could be not fully general in nature. Moreover the qualitative calculus, in a sense, propagates the approximation of the model through the intermediate results. As a consequence, the final results of qualitative and quantitative analysis could be different.

Qualitative physics is a physical theory, as the classical quantitative one, and is concerned with the general from of the laws by means of which any physical system can be described. The main difference between the two approaches is that in qualitative modeling the focus of attention is on how the modeled system achieves its behaviour, instead of on what this behaviour is. The quantitative descriptions give information on the parameters values at an instant. The qualitative descriptions give information about changes in the values and about the way the changes are related.
The traditional models are spatially unbounded, i.e. give information about any point in space; on the contrary the qualitative ones are spatially bounded. They can hence be used to describe the behaviour of a specified group of physical objects independently of the boundary conditions. This concept will become clearer as soon as some details on the current naive physics theories are provided.

The formal apparatus which must be provided by a qualitative theory is primarily concerned with behavioural issue, namely explanation and prediction.
From this point of view, a qualitative inferential appartus must suitably support a number of typical reasoning styles:

Determine activity
Deducing what is happening in the system at a given time. It is the ability to fully observe the state of the system as far as the possible incompleteness of the model allows. The relation between deepness of the observations and incompleteness of the model could be very interesting for an intelligent system that must show some judgement capability on its models.

Prediction
Deducing whath will happen in the future starting from some given situation.
Often multiple possible futures arise due to the incomplete knowledge embodied in the qualitative model. This is the inferential activity primarily involved in envisionment.

Postdiction
Deducing how a particular situation could have been achieved. It is a very difficult task because it is necessary to manage hypothesis generation.

Skeptical analysis
Determining the consistency of a description of a physical situation. It is a very important in evaluating the user's exploratory theories (for example in a ICAI system), and the ones which are generated by the system itself.

Measurement interpretation
Given a partial description of the components of the physical system and some observations of their behaviour, enhancing and enlarging the description of both the system and its behaviour [8].

Causal reasoning
As pointed out above, a central goal of qualitative modeling is to build description of physical systems which emphasize how the behaviour is achieved.
Thus qualitative description of system dynamics must be given in terms of related changes. Any change in a parameter value for a component must be related with some other changes in the components in its neighbourhood in the structure. In other words, any change must be attributed to the influence of a components and to other changes.
Causal reasoning is a crucial point in explanation generation. The main difficulty is that physical formal reasoning is often non-casual, especially when the reasoning activity of an expert applied physicist is considered. For example, the proofs carried out by reductio ab absurdum are non-casual in nature. Predictions are often based all-but-one-possibility exclusion, in a typical reductio ad absurdum style, and cannot be carried out in a fully causal fashion. This makes it a central point in naive physics research to link explanation and prediction.

3. THREE DIFFERENT APPROACHES

Since the late '70s a number of research groups have focused their studies on qualitative modeling. Their applicative goals are quite different, as well as the resulting theories. Nevertheless, there is a great information flow among them and their efforts are cooperative. In this section we briefly outline the main assumptions and the characteristics of the three most popular theories. This discussion is very general and is aimed at giving an overview of the main ideas. Some details on the most important issues in the three naive physics theories will be provided, in a synoptical way, in the next sections.

3.1. Qualitative Modeling Based on confluences

This research is carried on primarily by J. de Kleer and J.S. Brown [2], [5], [9]. The main interest is in explanation of behaviour. As a consequence casual reasoning is a central issue. The fundamental assumption is that complex physical systems can be modeled as structures composed of more elementary, relatively simple devices. The behaviour of the whole system is determined by the behaviour of its interconnected components. Models are seen as sets of descriptions of abstract communicating machines.

The proposed framework is shown in Figure 2. The physical systems is described in terms of the topology of its components. A component library contains the general models of a large class of devices that are typical of the application domain. For example, a pressure regulating valve is described in terms of a pressure sensor, a pressure regulator, a valve and a number of conduits. Component descriptions are expressed in a sort of qualitative differential equations called confluences; connections are modeled by sharing

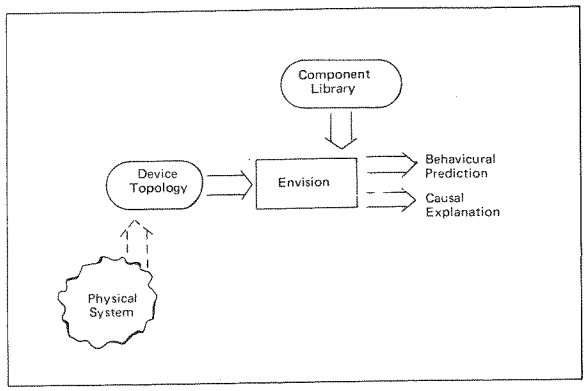

Fig. 2 Envision

variables between confluences belonging to different components. A general program (ENVISION), starting from the two above objects, carries out the qualitative simulation and produces behavioural prediction and casual explanation.

Inference about the system behaviour must be drawn starting only from the laws that rule the behaviour of its parts. As explanation must be casual, difficulties arise in using formal proofs constructed by using confluences as axioms of a logical theory. In fact, some useful inference rules in this kind of reasoning are based on reductio ab absurdum. From this point of view, there is a conflict between prediction and explanation reasoning tools. Predictions could be performed by means of a proving activity, but this kind of reasoning is not suitable for explanation purposes. The problem is solved by defining a new form of casual reasoning, based on strongly constrained local non-causal inferences, that seems to be good for both prediction and explanation.

Physicists work in two directions: classification and abstract description of a set of physical devices typical of a specified domain, and topological description of particular, application oriented-systems. It is mandatory to keep these descriptions indipendent. Giving a complete behavioural description of a device wothout considering a particular supporting context is generally a difficult task. Sometimes it can even be impossible. For example, a general description of a valve cannot specify what the input conduit is without violating the symmetrical nature of the flow in the conduit; but the behaviour of the valve as a whole can hardly be described without some indication about the input-output flow.

In order to overcome such difficulties some indications and principles are pointed out in this theory with the purpose of helping the descriptive effort. These principles are not a panacea, they rather give some hints in clearly guessing what is general and what is application oriented in a working description and, as a consequence, which assumptions are generally supported and which are strictly confined in a particular context.

No function in structure

Descriptions of general devices should not contain any assumption about their funcion, i.e. the goal in using them.

Class-wide assumptions

General descriptions must be partitioned in classes. The laws that are typical of a particular device must be clearly distinguished from those that are common to a class as a whole. It is clear to the authors, that the class-wided assumption theory is very difficult and largely unexplored and that it involves a number of fundamental epistemological topics.

Locality

The laws of a component cannot specifically refer to any othe part. A components can interact only with neighbouring components and neighbours must be identifiable in the structural description.

3.2 Qualitative process theory

This research is mainly due to K.D. Forbus [4], [7], [8]. The Theory is centered around the concept of change. the aim is to explicitly represent and manipulate the things that cause changes over time, namely the processes. All the intuitive ideas of dynamically related changes are modeled with a process, e.g. moving, cooling down, heating up, compressing, stretching.

A process explicity represents the relation between the changes and the information intuitively associated with them, e.g. when and why the process starts and stops, etc. A process is thus constitued by a description of objects involved by the process, by a set of preconditions which must be verified to make the process active and by a set of relations over the objects parameters that hold when the process is active. Moreover a process description contains information about the parameters which can change their value when the process is active. Processes only are responsible for all the changes in the system dynamics. Any change in a parameter value is, in fact, ascribable to an explicitly expressed influence of a process over that parameter. Processes represent, in a sense, the casual view of the physical system dynamics.

Processes have a central role in expressing the commonsense knowledge about the behaviour of physical systems and in computing prediction. The explanation of the behaviour of the system is carried out by means of histories. A history is a graph that describes the changes of a physical object in terms of the sequences of the values assumed by the object parameters. Processes can be included in histories in order to give casual information. An example of a history is presented in Figure 3.

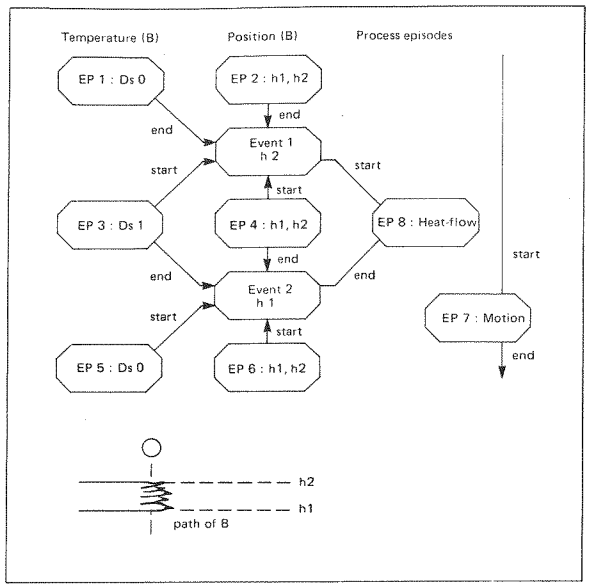

Fig. 3 A piece of the history for a ball dropping through a flame, with process episodes added. EP (n) are episodes, and time runs from top to bottom.

Due to the fundamental role of explicitly represented dynamical information, qualitative process theory is, in a sense, the most asymmetrical one with regard to the classical quantitative physics.

As the two above theories are centered in static structure and in dynamic evolution respectively, knowledge centralized by the first theory is distributed by the second and vice versa. Both these two naive physics theories propose a new language to express commonsense knowledge about systems behaviour, but the two approaches are quite different. A language must have primitives, methods for combining primitives and methods for creating new entities, abstracting combinations of primitives ad other abstractions. For example in LISP, primitives are car, cdr, cond and so on; they are combined mainly by nested functional application and abstractions are created by means of lambda expressions.

As pointed out above the proposal of de Kleer and Brown is to describe systems as machines. We then have elementary devices, structurally combined and possibly abstracted in devices classes. On the contrary in qualitative process theory processes could be the primitives of the language. A natural way of aggregating processes is suggested by groups of processes that share some components and sequences of processes occuring on the same component. A form of abstraction can be defined starting from the classification of processes.

In specific domains only processes of certain classes can occur. Examples of classes of processes in the motion domain are: flying, sliding, rolling, etc. Pro-

cess abstraction must have parametrized objects and it must contain relations and preconditions valid in general cases. For example a billiard game should be described with instantiation of two dimensional sliding processes, in which parametrized objects are filled with billiard balls.

3.3. Structural qualitative equations

This research is mainly carried on by B. Kuipers [3], [6]. His theory is actually a naive mathematics rather than a naive physics, as explicity declared by the author. Kuipers's theory seems to be the most well-established, though the most restricted one. The attention is focused on qualitative simulation and on defining a formal famework in which to perform prediction, envisionment and explanation of the system behaviour in a casual way. Models are created starting from classical quantitative analysis expressed in quantitative equations. No efforts is thus devoted to deriving laws from observations, i.e. to the physicist's task. Problems arising in laws producing are not addressed, thus a number of stimulating questions are left apart.

Qualitative models are structures which represent, in a qualitative way, the equations that describe the system. They are structural differential equations. It is important to note that structural models represent the equations and not directly the physical system; thus information about the system topology can be lost. This choice, on the contrary, implies that the "flow" of influences among the parameters values is explicity represented. In fact structural models are a kind of logical circuits where the wires represent the parameters and carry their values, and different kinds of logical operators represent the relations among the parameters values and trends.
Causal dependencies among values changes are structurally imbodied in the data structure which leads the predictive reasoning.

Qualitative simulation is performed by propagating qualitative values in the structural models and it is controlled by a well-stated semi-formal set of rules. When an ambiguous situation stops the information propagation in the casual structure, e.g. when an operator has incomplete input values, the structure is summarized up. This summarization produces a simplified causal structure which represents a set of equations which describes the system in a more stylized way. Qualitative simulation then proceeds in the new structure which should contain a lower number of ambiguities.

A central point in Kuiper's framework is the ability to guess previously unknown critical value points in the system behaviour, e.g. equilibrium, critical or oscillatory states.

4. CENTRAL TOPICS IN NAIVE PHYSICS THEORIES

In this section we point out some central issue of naive physics from the point of view of the three aforementioned theories. This presentation is aimed at providing a few pieces of synoptic information for a critical analysis. Our overview does not cover deep technical issue. Technicalities are crucial in a physical theory but cannot be summarized without loss of meaning; thus we keep this discussion informal and refer the reader to the original papers for technicalities. This section might contain some ideas already discussed in the previous one. The difficulty is that a synoptical presentation, though necessary, is not adequate by itself to give a clear synthetic idea of the single approaches. Every point is briefly introduced and the point of view of every qualitative theory is given.

4.1 Quantities.

The first point in which a model is qualitative in the definition and manipulation of parameters values. As naive physics emphasizes changes, quantities and value are considered with regard to their relations with changes. Hence a value is significant when it causes a change in the system if assumed by a parameter, or when it is reached as a consequence of a change. In this sense the number of values the physicist has to consider is finite and relatively small.

Thus, in a qualitative model, the parameters can have symbolic values in finite sets of quantities, i.e. qualitative discrete spaces. Due to this nature of the values of the parameters the derivates cannot be derived with ordinary differential calculus. A qualitative simulator has then to carry out a qualitative analysis for both parameters and derivates values.

All the considered theories use the notion of quantity space. This concept has been proposed by K.D. Forbus in [7]. The idea is that the qualitative discrete values for each parameter must be structured in a space. Quantity spaces are sets of simbolic values structured with a partial order; any values that can be assumed in sequence by the parameter are ordered.

In commonsense reasoning parameters are often considered only with respect to their positivity or negativity and to the positivity or negativity of their derivatives, i.e. their trends. Thus, in a great number of situations, the most important quantity space has only three values represented by +, – and 0. This is often all what is needed for casual reasoning.

Qualitative values are manipulated by means of a simple qualitative arithmetic which is used to compute the expression that represent the qualitative equations.

4.2 Statical description

Here we outline the approach of each theory to the problem of describing the physical system from a statical point of view. By statical description we mean the representation of the physical objects which constitute the system and their properties, as well as the structural dependencies among the objects.

- *Confluences*

System components are described in terms of confluences, a kind of qualitative differential equations. Each confluence belongs to a unique component description.
Confluences share parameters of the different objects which constitute the system; thus their topology and structure is modeled by variable sharing. A set of values can satisfy or contradict a confluence. If some parameters have no known value, i.e. a variable cannot be assigned a value, the confluence could be neither satisfied nor contradicted, it simply carries no information. This lack of information does not disturb the consistency of the model as a whole. In fact, in qualitative modeling, the knowledge about the system behaviour is produced during the solution process, i.e. qualitative simulation, and not reasoning on already computed solutions.

In quantitative physics a component behaviour can often be modeled by a single set of equations. In this case a notion of state or value region is usefull only for approximation purposes. In fact a partition of the possible parameters values is usually proposed when solutions of equational models are intuitively analyzed. This helps visualize and recognize commonsense ideas in the abstract description of the system behaviour provided by the mathematical solutions. On the contrary, a single set of confluences cannot fully describe the components behaviour. In fact qualitative equations are intrinsecally approximated and, in particular ranges of variables values, lose their physical sense.

Usually the nearer a confluence is to the commonsense description of a relation, the more this problem arises. It is necessary to define qualitative states for any component, depending on qualitative regions in parameters values. For each qualitative state, the model must provide the proper confluences.

The most used example is the description of a valve. It has three qualitative states: OPEN, CLOSED and WORKING. Such states are characterized by the value of the area (A) through which the fluid flows: respectively A=MAX, 0‹A‹MAX, 0‹A‹MAX and A=0. The complete set of confluences in this small model is reported in Figure 4.

A model is said to be pure if the confluences of each of its states relate either parameter values only or derivatives only. An impure model can be reduced to a pure one by defining a proper number of states. As parameters and derivatives need to be analyzed in a homogeneous way, it is then clearly useful to distinguish them by means of separated confluences.

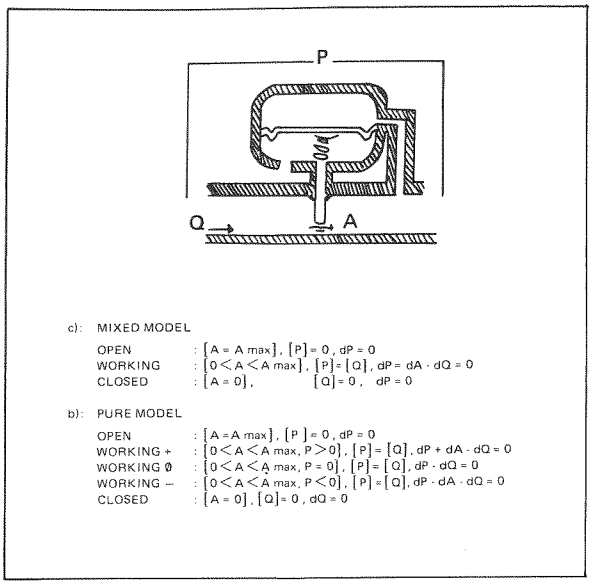

Fig. 4 Pressure-regulator

- *Qualitative process theory*

In this framework, descriptions are centered in related changes and their causal ordering. Statical descriptions are thus instrumental to the dynamical ones. They are useful to describe strong relations among elementary physical objects which hold in particular instant; in a sense statical descriptions are tool to clarify some particular physical situation with respect to a defined time interval, and to manipulate composite objects in a natural way. In particular cases the objects which are involved in commonsense are constitued by a set of more elementary physical entities, strongly related to one another. For example a filled bottle is constituted by a particular container and a certain amount of contained liquid. In a defined time interval the bottle can be empty and the proper description of the whole object is different.

In qualitative process theory statical situations are described by means of individual views. An individual view is constituted by a description of the elementary objects with their internal parameters, a number of preconditions which must be verified for the view to be active, and a set of expression which create relationships among the objects and which hold when the view is active. The relation among the elementary objects are expressed in terms of an interesting new notion of qualitative proportionality among their parameters values. The intuitive seman-

tics of this proportionality is integrated in the proposed qualitative simulator and hence in the specific qualitative arithmetic.

Individual views are statical descriptions, but they can be associated to dynamically represent the behaviour of the physical system. In fact a sequence of different views that involve the same elementary objects describes the evolution of a physical situation along time. This can be useful for clearness in particular descriptions, but it does not match the causality requirement for a qualitative dynamical model.

- *Structural qualitative equations*

The qualitative equations which describe the physical system are represented by a graph. Arcs represent variables and nodes represent the operators that connect variables and constants in the equations. Three types of operator are allowed: arithmetic, functional and derivates. Arithmetic operators represent expressions of the form Y = X + Z, where X and Z are variables or constants. They are the ordinary arithmetical operators and refer to a proper qualitative arithmetic. Functional operators are of the form Y = M + M + (Y), where M + (Y) is a stricly increasing function of X (decreasing M-); they describe any functional non-arithmetic dependence among parameters.
Derivates operators are or the form Y = dX/dt.

Causal structures look like circuits, as already pointed out, and are built following a well-stated set of rules.

4.3 Dynamical description and predicton

By dynamical description we mean two things. The first is the qualitative simulation mechanism, and the second is the way to embody commonsense knowledge about the behaviour of the system in the description of a physical system. The former is actually naive mathematics, the latter is a very important feature of the qualitative modeling idea, whose lack is critical in quantitative modeling.

- *Confluences*

This theory does not provide a tool to embody dynamical information in the models. The central idea is, in fact, to derive qualitative description directly from the structure of the system, building a library of component model incrementally. This approach is thus aimed at exploiting available experience in classical modeling with the approach of the classical systems theory.

As statical descriptions are decomposed into components and components into qualitative states, a qualitative simulator must deal with such partitioned models. In fact it has to devise and manage both intrastate and interstate behaviour of the model.
At some instant, each component of the model is in a qualitative state that determines which confluence can properly model its behaviour. The qualitative state of the system can be defined as the union of the qualitative states of each component.
Intrastate behaviour is the behaviour of the system when it is in a defined qualitative state, i.e. when is modeled by a set of confluences that is the union of the proper confluences of each component in its own qualitative state.

In a single system qualitative state, confluences are like a set of contraints on variables; intrastate behaviour is thus determine by a constraint propagation activity. The propagation of the constraints is carried out exploiting the results of a recent MIT research [11]. It is clear that clearer results can be obtained from pure models. Actually the confluence process takes place by interleaving constraint propagation and generate and test techniques in order to overcome the typical difficulties arising in resolving systems of equations by constraints propagation. Contraints propagating can in fact be considered as solving systems of equations with the substitution method. The process stops when a set of qualitative values for the variables is reached, which makes all the proper confluences satisfied. It must be noted that qualitative calculus does not satisfy the field axioms; hence multiple solutions can result even if the number of equations is equal to the number of unknowns.

When a parameter has a value that violates the qualitative state conditions of the current state of the component, the state of the model and the set of proper confluences must be changed. For example when the flow area of a valve vanishes the valve changes to CLOSED state so that it is described by a differential set of confluences. When a component changes its qualitative state, the system qualitative state changes too. Interstate behaviour is the transition between qualitative states.
The qualitative simulator is able to devise when the system changes its state and to draw a graphical picture of the possible state transitions, starting from a defined situation. This process is performed by means of a provided algorithm which is based on considering parameter values trends and possible conditions violations.

In such a way system is able, following a stated set of strategical rules, to perform an envisioning activity. Indeed a complete state diagram representing all the possible transitions can be produced.

- *Qualitative process theory*

Dynamics is the central topic in this theory. Thus

great attention is devoted to dynamical and causal knowledge representation. Processes are the way dynamical information is embodied in descriptions. Processes represent the intuitive idea of physical communication; they explicitly refer to the mass and energy exchanges that are hidden in equational quantitative models. A paradigmatical example of this concept is given by the laws of conservation: this kind of laws assert that, in insolated systems, the total amount of a defined quantity is constant. This representation does not contain any information about the mutual exchanges among the physical components of the system that take place in order to satisfy the law. In a few words, processes can express the arguments that humans use to explain the solutions of quantitative physics.

It is clear that qualitative simulation must be strictly linked to the process structure. The central assumption of the theory is that all changes in the physical system are caused, directly or indirectly, by processes. Behavioural analysis of systems descriptions in a given physical situation can thus be performed, in an iterative way, as follows:

. . all possible processes are recognized by objects grouping, i.e., all the possible sets of objects that can be involved by a process are determined scanning the objects descriptions of all the process descriptions;

. . all the active processes are determined evaluating the preconditions of the process descriptions and possibly performing a search in some quantity space in order to manage incomplete information;

. . all the changes in the system parameters are guessed by computing all the influences exerted by the active processes on each parameters. Changes can influences the set active processes which must be recomputed according to a small set of intuitive rules.

Times is explicitly represented in the framework in terms of intervals, as suggested by J. Allen [12] (see the proper section of this document). Actually this time representation theory seems to be the most suitable for qualitative physics in general; the notion of intervall is more intuitive than the concept of state in situation calculus, for example.

A qualitative simulator is able to present a synthetic view of what is happening in the system evolution; for each simulation step, it computes all the active individual views in a way similar to the active processes. These give a description of the situation in terms of the intuitive objects aggregates.

• *Structural qualitative equations*

Great care is devoted to giving a well-stated set of rules for qualitative simulation.

As causal structures look like circuits, and hence like networks of constraints on variables, behavioural analysis is performed by means of constraint propagation techniques [11]. Kuiper's qualitative simulator has, as a consequence, the same problems as de Kleer's one in dealing with deadlock situations. When such a case occurs, and a simple generate and test method is not affordable, this theory provides a very interesting set of rules that allows to summarize the physical system description. The new description represents a set of more approximate equations, a weaker model in a sense. The new causal structure, being more stylized, should not contain the conflicts and the ambiguities that caused the trouble. Working in this environment it is then possible to proceed in a significant, though less accurate, behavioural analysis overcoming computational difficulties.

Another set of rules is provided that make the simulator able to devise previously unknown significant states in the system dynamics in terms of the corresponding values for the variables. For example, equilibrium states, oscillations and critical trend inversions are determined.

4.4 Explanation

The explanation issues are the most asymmetrical with respect to classical quantitative physics. An explanation must be concerned with the way the system achieves its behaviour rather than simply with what the behaviour is. This means that explanation is related to causal reasoning which is the central point of naive physics. In fact a basic assumption in qualitative modeling is that in commonsense reasoning what is necessary is to relate changes to each other, while remarkable quantitative precision in the conclusions is not so important. The explanation capability of qualitative modeling framework is the ability to devise and to exploit all causal knowledge explicity or implicity involved in the model of the physical system.

• *Confluences*

As models are oriented towards the structure of the system rather that towards its dynamics, the explanation and the involved causality are a critical point and great attention is devoted to it. A good explanation for the behaviour of a modeled system must be testable, compelling and succinct. Testable means that it must be possible to test its validity syntactically; the explanation must thus contain all relevant information. An explanation is compelling when it constitues a proof for its conclusions.

Explanations are also important for communicating commonsense knowledge and must then be succint. Obviously all these requirements must be met while speaking about the way the system behaves in a cau-

Confluences can be seen as axioms expressed in equational form in a logical framework. In a sense, a qualitative simulator determines the intrastate behaviour choosing and applying the proper axioms. As a consequence it perform a proving activity in a simple logical framework with simple inference rules. A trace of the actions and the result of the simulator can be a kind of informal proof and a explanation. Formal proofs meet the above requirements very well: they are compelling, testable and succinct by definition. The problem is that they are not causal. If, in deducing a possible parameter value, a reductio ab absurdum method must be used, the proof can explain what the system behaviour is, i.e. which states the system can reach. However, it gives no information about the way these states are reached. As indirect proofs are a kind of reasoning by exclusions, changes are not attributed to any componente in particular.

In order to exploit the good properties of formal proofs, a deep analysis is conducted about the relation arising between proof techniques and causality. This study results in defining a new notion of causality and time, the so called mythical causality and mythical time; these can be seen as representations of the concepts of time and of sequence of related changes in proof and explanation as opposed to time and changes in the actual physical system. In a sense it is an effort aimed at limiting the indirectness of the produced proofs by distinguishing proving methods and generated explanations. With this idea as a basis, a set of heuristic rules is proposed in order to generate causal proofs to be used as explanations.

- *Qualitative process theory*

As processes are responsible for all the changes in the system model parameters, the dynamic description of the behaviour of the physical system in terms of active processes and their influences on parameters values constitutes an explanation by itself. Such explanations are, in fact, causal by definition. In particural active processes and views give a commonsense causal description of the system as could be frozen in a photograph.

Another explanation tool si provided with the histories generation capability. The qualitative simulator is able to produce a kind of synoptical analysis of parameters values our time. For each parameter is generates a description of all the values the parameter has taken and of the related time intervals. Every such stripe contains all the information about what values the parameters has had and when. The single stripes are connected among them according to the partial ordering of the time intervals involved.

- *Structural qualitative equations*

Causal relations among the parameters changes are embedded in the descriptive structure. Actually the structure represents the connections among the operators that describe the causes of changes. Explanation is carried out drawing the constraint propagation and the satisfaction flow through the causal structure, i.e. by tracing the activity of the qualitative simulator. Explanations are thus causal by definition. Some interesting additive information can be extracted from the summarization process and from the behaviour of the summarized descriptions.

4.5 Laws extraction

A physical theory must provide a framework and some methodology to extract behavioural laws from observations of a physical system. Kuiper's theory does not address this important question and derives its laws from the quantitative analysis with a simple method. The other theories provide a clear methodology for qualitative modeling based on both commonsense knowledge embodying and quantitative laws translation. Confluence theory seems to suggest a method of analysis quite similar to the quantitative modeling one: decomposition and formalization in equational terms are very popular tasks in traditional physics.

From this point of view, qualitative process theory could seem to adopt an unusual and quite painful method for extracting descriptions: the user must explicitly define each of the process involved in the system behaviour. It must be noted, nevertheless, that commonsense knowledge is concerned with qualitative descriptions of concepts that are easily expressed with processes. This keeps qualitative reasoning apart from classical modeling and makes it easier to explicitly include causal information about processes in models. This point seems to be confirmed by an analysis of the examples developed by the author and reported in the papers on qualitative process theory.

5. ACKNOWLEDGEMENT

This analysis was carried out in collaboration with CISE, Milano. I would thank Massimo Gallanti and Alberto Stefanini for the critical discussion which have been crucial for the refinement of this paper.

6. REFERENCES

[1] P.J. Hayes, THE NAIVE PHYSICS MANIFESTO, in D. Michie (ed.), "Expert Systems in the Microelectronic Age", Edimburg Univ. Press, Edimburgh, 1979.

[2] J. De Kleer and J.S. Brown, THE ORIGIN, FORM AND LOGIC QUALITATIVE PHYSICAL LAWS, Proc. 8th Int. Joint Conf. on Artificial Intelligence, Karlsruhe, FRG, 1983, 1158-1169.

[3] B. Kuipers, GETTING THE ENVISIONMENT RIGHT, Proc. AAAI-82, Pittsburg PA, 1982, 209-212.

[4] K.D. Forbus, QUALITATIVE REASONING ABOUT PHYSICAL PROCESSES, Proc. 7th Int. Joint Conf. On Artificial Intelligence, Vancouver Canada, 1981, 326-330.

[5] J. De Kleer and J.S. Brown, A QUALITATIVE PHYSICS BASED ON CONFLUENCES, Artificial Intelligence 24, 1984, 7-83.

[6] B. Kuipers, Commonsense Reasoning About Causality: DERIVING BEHAVIOUR FROM STRUCTURE, Artificial Intelligence 24, 1984, 169-203.

[7] K.D. Forbus, QUALITATIVE PROCESS THEORY, Artificial Intelligence 24, 1984, 85-168.

[8] K.D. Forbus, MEASUREMENTS INTERPRETATION IN QUALITATIVE PROCESS THEORY, Proc. 8th Int. Joint Conf. On Artificial Intelligence, Karlsruhe, FRG, 1983, 312-320.

[9] J. De Kleer and J.S. Brown, QUALITATIVE REASONING IN HIGHER ORDER DERIVATES, Proc. AAAI-84, Austin TX, 1984, 86-91.

[10] D.G. Bobrow, QUALITATIVE REASONING ABOUT PHYSICAL SYSTEM, Artificial Intelligence 24, 1984, 1-5.

[11] G.J. Sussman and G.L. Steele Jr., CONSTRAINTS - A LANGUAGE FOR EXPRESSING ALMOST HIERARCHICAL DESCRIPTIONS, Artificial Intelligence 14, 1980, 1-39.

[12] J. Allen, TOWARDS A GENERAL THEORY OF ACTION AND TIME, ARTIFICIAL INTELLIGENCE n° 23, 1984, pp. 123-153.

COMPUTAZIONE PARALLELA: UNA PANORAMICA

F. De Santis, A. Negro, E. Pisano
Dipartimento di Informatica ed Applicazioni - Università di Salerno

RIASSUNTO

Il crescente interesse rivolto dagli studiosi alla computazione parallela nel decennio trascorso è ovviamente motivato dalla necessità di abbassare i limiti della complessità di tempo e di spazio.
Il problema della parallelizzazione di processi computazionali è stato affrontato su differenti livelli: sono stati studiati modelli teorici di computazione, specifici linguaggi di programmazione, implementazione hardware e organizzazione di dati. Qui vengono presentati alcuni degli aspetti fondamentali che riguardano le computazioni parallele con lo scopo di porre l'attenzione sui maggiori problemi che devono essere risolti nel disegno di algoritmi paralleli.

ABSTRACT

The increasing interest devoted by scientists to parallel computation in the past decade is obviously motivated by the need of lowering space and time complexity bounds.
The problem of paralleling computational processes has been approached at different levels: theoretical computation models, specific programming languages, machine hardware implementation, data organization and access have been studied.
Here we survey some of the fundamental aspects concerned with parallel computations with the aim of emphasizing the major problems which are to be solved when designing parallel algorithms.

1. INTRODUZIONE

L'interesse sempre crescente dimostrato negli ultimi venti anni verso la computazione parallela è motivato dall'esigenza di abbassare i limiti di complessità di tempo degli algoritmi preposti alla soluzione di classi di problemi. D'altro canto, la produzione di componenti hardware sempre più veloci e sofisticati e la messa a punto di architetture sempre più flessibili hanno, a loro volta, dato ulteriori stimoli alla ricerca nel campo dell'elaborazione parallela.

Il problema della parallelizzazione dei processi computazionali è stato affrontato in diverse forme ed a diversi livelli. Sono stati studiati linguaggi dotati di strutture progettate esplicitamente per rappresentare il parallelismo, modelli per la computazione parallela, reti di processori realizzabili direttamente in hardware, tecniche per individuare e misurare la quantità di parallelismo intrinseca di un algoritmo.

Tuttavia, due sembrano essere gli approcchi più interessanti: da un lato si punta allo sviluppo di metodologie che consentano di progettare direttamente algoritmi paralleli per risolvere specifiche classi di problemi anzichè parallelizzare algoritmi sequenziali esistenti; dall'altro si cerca di definire modelli astratti di macchine parallele, studiandone le capacità computazionali e si indaga sulla possibilità di darne delle realizzazioni pratiche.

Tra i problemi principali studiati, vi sono da un lato la individuazione delle fasi di elaborazione indipendenti di un algoritmo e la valutazione dei vantaggi derivanti dalla messa a punto di una macchina ad hoc per la sua computazione, dall'altro l'analisi della validità e consistenza algoritmica di una data struttura parallela. In ogni caso, si segue la filosofia secondo cui un algoritmo ben si adatta ad una certa carta geometria di comunicazione solo se questa è sfruttata efficientemente, ovvero solo se la complessità ottimale dell'algoritmo non può essere ottenuta facendo uso di una macchina più semplice.

In questo lavoro analizzeremo i principali aspetti delle problematiche connesse allo sviluppo di algoritmi paralleli senza, tuttavia, scendere nei dettagli di algoritmi specifici se non, a titolo puramente esemplificativo, nell'ultimo paragrafo.
Evidenzieremo, in primo luogo, le correlazioni tra tipo di achitettura e sviluppo degli algoritmi, sottolineando, in particolare, le peculiarità dell'architettura sistolica.
L'analisi del concetto di algoritmo parallelo e delle relative proprietà porterà, in maniera naturale, a cercare di definire parametri significativi per la valutazione delle prestazioni dell'algoritmo stesso ed a chiedersi in quale modo possano essere efficacemente valutati i vantaggi derivanti dal processo di parallelizzazione di un algoritmo.

2. ARCHITETTURE DI ELABORAZIONE

Gran parte del software parallelo è stato progettato pensando ad architetture SIMD e MIMD, ovvero, in accordo alla classificazione di Flynn [14] dei sistemi di calcolo, ad architetture che possono trattare un solo flusso di istruzioni e più flussi di dati o più flussi di istruzioni e più flussi di dati.

Una macchina ad architettura SIMD (macchina SIMD) è tipicamente costituita da un'unica unità di controllo, N unità aritmetico-logiche (ALU) sincrone, N banchi di memoria, un insieme di funzioni $F = \{f_1, f_2 \ldots f_n\}$ di interconnessione ed uno schema di mascheramento per il controllo dello stato delle ALU. L'unità di controllo può accedere ad uno qualsiasi dei banchi di memoria, i quali, però, sono in corrispondenza uno a uno con le ALU. Una ALU può operare su dati di un banco di memoria che non sia di sua competenza solo attraverso un registro speciale, il DTR (Data Transfer Register): la i-esima ALU, $1 \leq i \leq N$, può trasferire il contenuto del suo DTR nel DTR della f(i)-esima ALU con $f \in F$.

Una macchina ad architettura MIMD (macchina MIMD) è, generalmente, un insieme di macchine general-purpose asincrone, ciascuna in grado di eseguire un flusso di istruzioni indipendentemente dalle altre e con queste comunicante al fine di cooperare alla soluzione di un unico problema. Un insieme di macchine SIMD, connesse mediante una opportuna rete di comunicazione, realizza, ad esempio, una macchina MIMD.

Da quanto detto, appare chiaro che le macchine SIMD sono particolarmente adatte alla soluzione di quei problemi che richiedono poche operazioni da eseguire su dati diversi, mentre le macchine MIMD sono efficacemente utilizzabili per la soluzione di quei problemi che richiedono molte operazioni indipendenti, ed eventualmente distinte, da eseguire sugli stessi dati o dati diversi. In entrambi i casi, comunque, la descrizione di un algoritmo può essere fatta prescindendo dalle caratteristiche particolari della macchina da cui esso verrà eseguito e specificando, solo, le risorse richieste per un corretto funzionamento dell'algoritmo stesso.

Concettualmente molto diversa è l'idea alla base dell'architettura sistolica [1], [2], [3] che, proposta inizialmente per il progetto di circuiti VLSI, è diventata in poco tempo uno degli strumenti di base per la progettazione di algoritmi paralleli.
La ragione di tale successo risiede nel fatto che essa garantisce un alto grado di parallelismo sia per ciò che concerne l'elaborazione vera e propria sia per ciò che concerne l'input/output e la comunicazione tra le unità di computazione.

La realizzazione hardware di una macchina ad architettura sistolica (macchina sistolica), infatti, è costituita da un reticolo di processori semplici e primitivi attraverso cui i dati circolano in modo regolare. I processori, cui è demandata la funzione di effettuare istante per istante una ben determinata computazione, spingono i dati attraverso il reticolo così che questo sia attraversato da un flusso di dati costante: cruciale, ai fini del rendimento ottimale, è la geometria delle comunicazioni tra i processori. Per ciascuno di essi il controllo della computazione e delle comunicazioni con i processori adiacenti è semplice e l'occupazione di memoria è una costante molto piccola, indipendentemente dalla taglia della rete [12].

Contrariamente a quanto osservato per le macchine SIMD e MIMD, non è possibile descrivere un algoritmo prescindendo dalla descrizione dettagliata della macchina sistolica per la quale esso è stato progettato, in quanto la progettazione dell'algoritmo influenza fortemente la progettazione della rete determinandone le prestazioni ed i costi di realizzazione e di layout.

I problemi che ben si prestano a soluzioni con macchine sistoliche corrispondono ad algoritmi che:

i) richiedono pochi tipi differenti di processori elementari;

ii) consentono un flusso di dati semplice e regolare, ovvero richiedono che i processori comunichino mediante un reticolo semplice e regolare;

iii) permettono l'uso del pipeling e del multiprocessing.

Tali caratteristiche sono di importanza notevole nella progettazione di reti VLSI, usate sia come periferiche di calcolatori ospiti sia come componenti di sistemi special-purpose più ampi, al fine di mantenere un buon rapporto costo/prestazioni. Infatti da tale metodologia di progettazione segue che:

i') è possibile progettare e testare solo pochi processori elementari dal momento che la maggior parte dei processori presenti sul chip sono copie di pochi processori di base;

ii') la regolarità delle intercomunicazioni tra i processori implica che la progettazione può avvenire in maniera modulare;

iii') l'uso del pipeling e del multiprocessing assicura che le prestazioni di un chip possano essere ottenute semplicemente includendo molti processori identici sul chip.

È chiaro che, risultando la progettazione LSI di sistemi complessi particolarmente onerosa sia per la difficoltà della progettazione di ciascun tipo sia per il numero di diversi chips che dovono essere costruiti,

modularità, regolarità delle interconnessione e pipelining non solo riducono drasticamente entrambi questi fattori di costo, ma semplificano anche il layout dei singoli chips e del sistema nel suo complesso.

Nell'ultimo paragrafo di questo lavoro, riportiamo, a titolo di esempio, gli algoritmi sistolici e le relative reti VLSI che computano il prodotto di una matrice per un vettore ed il prodotto tra due matrici.

3. LO SPAZIO TRIDIMENSIONALE DEGLI ALGORITMI PARALLELI

Un algoritmo parallelo può essere pensato come un insieme di moduli indipendenti eseguibili contemporaneamente e mutuamene comunicanti; in [1], Kung individua tre parametri, o dimensioni, rispetto a cui classificare gli algoritmi paralleli: la granularità modulare, il controllo della concorrenza e la geometria delle comunicazioni.

La granularità modulare rappresenta l'ammontare massimo di computazione che un certo modulo può eseguire prima di essere costretto a comunicare con un altro: esprime, cioè, la maggiore o minore tendenza dell'algoritmo ad avere comunicazioni intensive tra i suoi moduli. Più precisamente, un algoritmo parallelo con piccola granularità modulare richiede frequenti comunicazioni intermodulari e viceversa.

Il controllo della concorrenza assicura la correttezza delle esecuzioni concorrenti, nel senso che gestisce le interazioni tra i moduli, così che l'intera esecuzione dell'algoritmo possa procedere in maniera corretta.

La geometria delle comunicazioni, infine, stabilisce la struttura geometrica della rete di processori.

Per gli algoritmi realizzati su macchine SIMD (algoritmi SIMD) e per gli algoritmi sistolici la granularità modulare è la stessa per ciascun modulo: in particolare, si mantiene costante per gli algoritmi sistolici ed aumenta all'aumentare del flusso dei dati per gli algoritmi SIMD.

Per ciò che riguarda il controllo della concorrenza è opportuno notare che nelle macchine SIMD non sussistono problemi di sincronizzazione, mentre nelle macchine MIMD la sincronizzazione fra i diversi processori cooperanti viene gestita mediante "semafori" sulle celle di memoria.

Nelle macchine sistoliche il controllo viene effettuato da semplici meccanismi locali e le comunicazioni sono imposte dalla organizzazione geometrica del reticolo di processori.
La geometria risulta tanto più conveniente quanto più è semplice e regolare, ovvero quanto più le connessioni tra i processori sono simmetriche e della stessa lunghezza. Ciò, infatti, assicura alta densità di processori su un singolo chip e massima flessibiltà di input/output per i dati. Gli array di processori linearmente, rettangolarmente ed esagonalmente connessi della Fig. 1 costituiscono esempi di semplicità e regolarità.

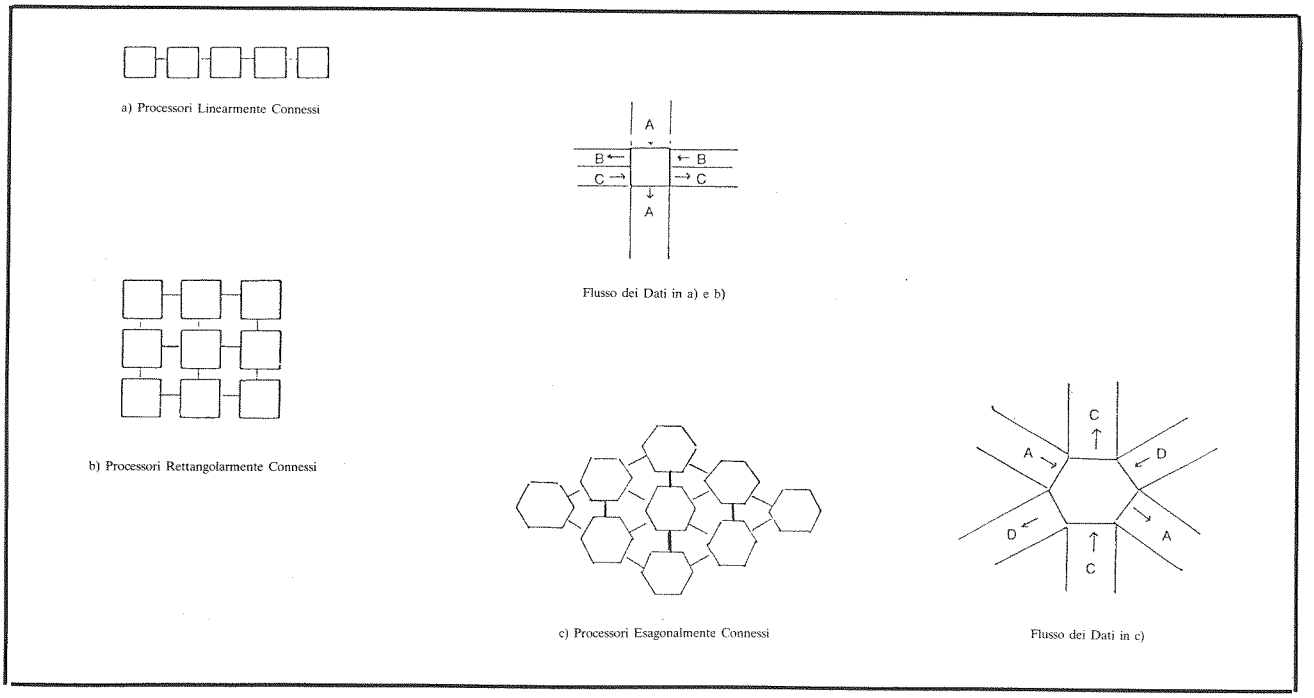

Fig. 1

4. MISURE DI EFFICIENZA PER ALGORITMI PARALLELI

Per ciò che concerne la valutazione della complessità degli algoritmi paralleli [4], è possibile definire la complessità in tempo T come per gli algoritmi seriali. Questa però dipende anche dal livello di parallelizzazione che, a sua volta, è legato (sempre nei limiti delle possibilità di parallelizzazione consentite dalla natura del problema) al numero di processori utilizzati ed al numero di linee di comuniazione necessarie per ogni processore. Nei circuiti VLSI, che devono essere realizzati nel piano, al crescere del numero di processori e di collegamenti cresce l'area necessaria per contenerli, A, che è quindi un parametro importante.

Uno dei problemi che si pongono è quello di introdurre un unico parametro che tenga conto contemporaneamente di questi due elementi fra loro interdipendenti, tempo ed area. Una delle scelte più comuni per misurare la complessità degli algoritmi sistolici risulta essere la misura AT^2, [5], [6], che può essere giustificata con il seguente esempio.
Supponiamo che i processori e i fili che li connettono siano contenuti in un quadrato di lato L, per cui $A = L^2$. Prendiamo ora una retta parallela a due lati del quadrato che intersechi gli altri due e sia ω il numero di fili tagliati da questa retta e λ il massimo tra il diametro di un filo e la distanza tra due fili adiacenti. Si ha

$$L \geq 2 \lambda \omega$$

La retta determina una partizione dell'insieme P dei processori in due sottosistemi disgiunti, X e Y.
Se I (X, Y) è il numero totale dei segnali che attraversano la retta durante una computazione, per andare da X a Y, e τ è il tempo di clock, si ha ovviamente

$$T \geq \frac{I(X, Y)}{\omega}$$

ove T è il tempo di computazione totale.

Sostituendo opportunamente, segue

$$A = L^2 \geq 4 \lambda^2 \omega^2 \geq 4 \lambda^2 \cdot \frac{I^2(X, Y)}{T^2} \cdot \tau^2$$

ovvero

$$AT^2 \geq K \cdot I^2 (X, Y)$$

ove la costante K ingloba le costanti tecnologiche λ e τ.
Il secondo membro di quest'ultima disegualianza viene a dipendere da quantità legate solo al problema, cosicchè la misura AT^2 rappresenta una proprietà naturale degli algoritmi sistolici, essendo, in particolare, una funzione del flusso di informazione che attraversa il reticolo.

Osserviamo, infine, che l'analisi dell'efficienza di un algoritmo parallelo risulta completa solo quando è stato effettuato il confronto con l'efficienza del corrispondente algoritmo seriale: particolare interesse assume, in questo senso, la conoscenza dello "speed-up-ratio", S_N, [7], ovvero del rapporto tra il tempo impiegato in seriale ed il tempo impiegato in parallelo per l'esecuzione di un dato algoritmo.
Certi algoritmi, infatti, sono naturalmente decomponibili in numerosi moduli indipendenti e, quindi, facilmente eseguibili in parallelo; altri, invece, sembrano richiedere una esecuzione intrinsecamente seriale.

Lo speed-up ratio dà, così, una misura della qualità di parallelismo insita in un algoritmo fornendo informazioni sulla effettiva convenienza del processo di parallelizzazione. Esso è formalmente definito come:

$$S_N = \frac{T_1}{T_N} \geq 1$$

dove T_1 è il tempo richiesto per effettuare una computazione utilizzando un solo processore e T_N è il tempo richiesto per effettuare la stessa computazione utilzzando N processori; T_1 e T_N sono misurati in unità di passi.
È chiaro che $\max S_N = N$, in quanto esso corrisponde alla condizione ottimale per cui una computazione, che in seriale richiede N passi, richiede in parallelo un unico passo. Lo speed-up ratio ottimale è, per ovvi motivi, difficilmente raggiungibile.

Vengono, comunque, considerati efficienti quegli algoritmi per cui S_N risulta pari a KN, dove K è una costante strettamente minore dell'unità: tanto più K è vicina ad uno, tanto più conveniente è il processo di parallelizzazione. Ancora efficienti sono considerati gli algoritmi per cui $S_N = KN/\log N$, in quanto la velocità di computazione è proporzionale al numero dei processori. Meno conveniente, invece, è la parallelizzazione di quegli algoritmi per cui $S_N = K \log N$, poichè la velocità di computazione incrementa di un fattore solo additivo quando il numero dei processori aumenta.

Del tutto inutile risulta la parallelizzazione dell'algoritmo quando S_N è indipendentemente da N, ovvero quando $S_N = K$.

Il grado di efficienza, E_N, di una prestazione parallela è, infine, misurato dal rapporto tra lo speed-up ratio attuale e quello ottimale:

$$E_N = \frac{S}{\max S_N} = \frac{T_1}{N T_N} \leq 1$$

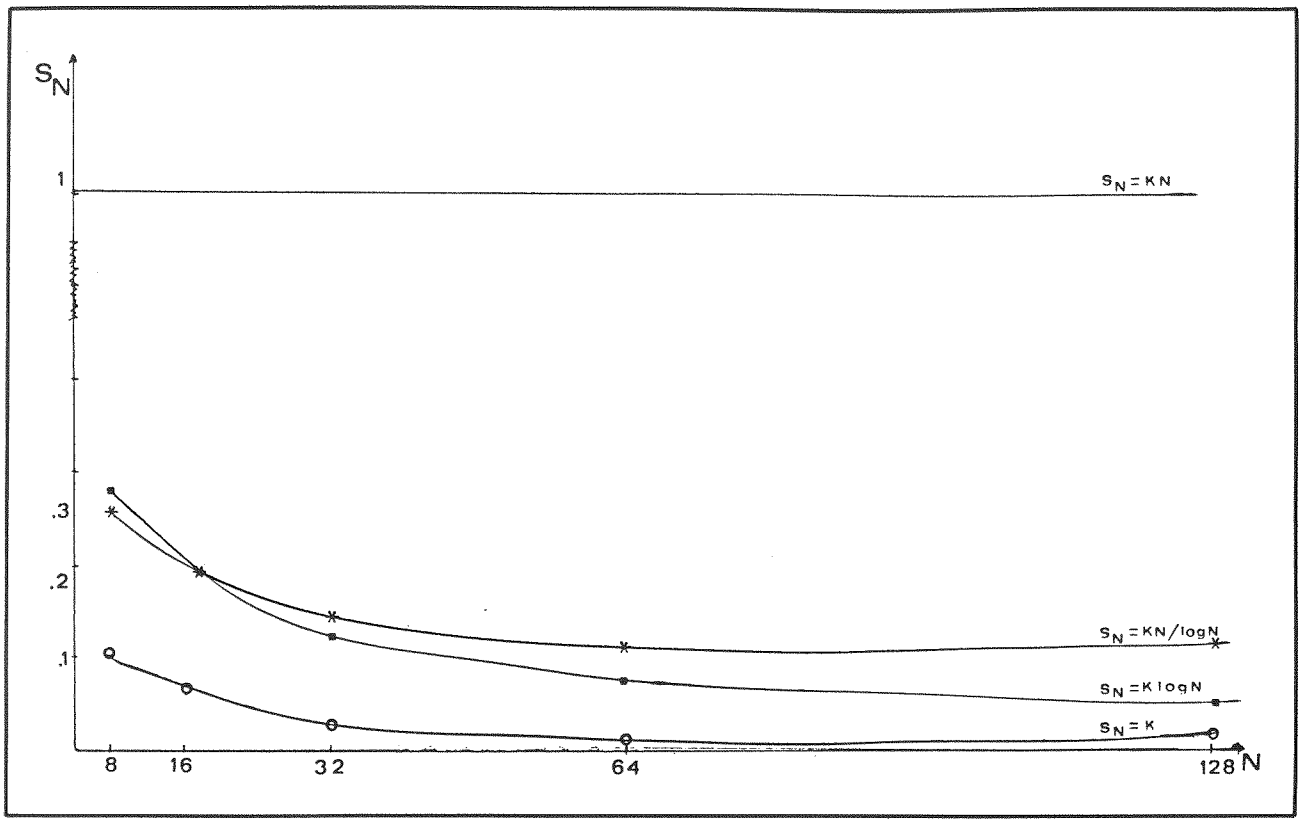

Fig. 2 - Speed-up ratio.

Nel grafico di Fig. 2 sono rappresentati gli E_N corrispondenti agli speed-up ratio appena analizzati per K= 0.9.
In accordo a quanto detto, risulta evidente che l'efficienza di una prestazione parallela sarà tanto più alta quanto più lentamente E_N decresce al crescere di N.

5. ALGORITMI SU MATRICI

Le computazioni su matrici hanno largamente beneficiato degli algoritmi sistolici: come, infatti, vedremo la complessità di tempo per tutte le operazioni tipiche tra matrici si riduce ad ordini di grandezza lineari. Gli stessi algoritmi sono, comunque, realizzabili anche su macchine SIMD con simili strutture interconnettive, il viceversa non è sempre vero, perchè le macchine SIMD hanno la capacità di memorizzare un ammontare relativamente grande di dati all'interno di ciascun processore ed, in quanche caso, l'assenza di tale capacità è cruciale ai fini dell'efficienza dell'algoritmo.

Particolarmente adatte ad operazioni tra matrici risultano essere le geometrie lineari ed esagonali i cui processori eseguono tutti la stessa istruzione di prodotto interno, C:=C+A.B [12], [13]. Ogni processore ha tre registri, R_A, R_B ed R_C, ciascuno dei quali ha due connessioni, una per l'input ed una per l'output (Fig. 3).

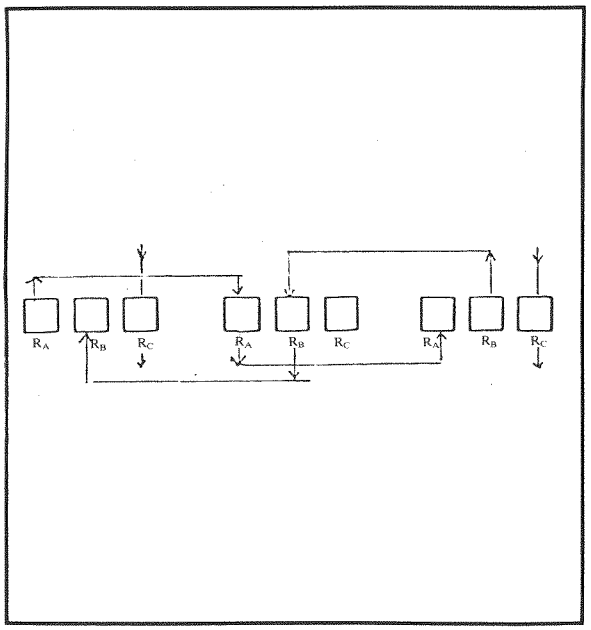

Fig. 3 - Connessioni tra i registri in una geometria lineare.

Al generico tempo t, un processore sposta i suoi input in R_A, R_B ed R_C e computa $R_C:=R_C+R_A \cdot R_B$; al tempo t+1 il nuovo valore di R_C, insieme con i valori input per R_A ed R_B, sarà disponibile in output.

5.1. Prodotto fra Matrice e Vettore

Si consideri una matrice quadrata di ordine n, A, ed un vettore $x= (x_1, x_2 \ldots x_n)^T$, dove T indica l'operazione di trasposizione; si voglia computare Ax=y.
La progettazione dell'algoritmo sistolico parte dalla considerazione che i prodotti interni parziali sono computabili indipendentemente l'uno dall'altro e "canalizzazioni" in modo tale che una loro opportuna sequenza vada a costituire le componenti risultanti. Un array di processori linearmente connessi realizza efficientemente questa idea. La matrice A, ruotata di 45° in senso orario, viene memorizzata in (2n–1) unità di memoria ed attraversa l'array dall'alto verso il basso; le componenti di x ed y (dove y è il vettore dei prodotti interni) attraverso l'array rispettivamente da sinistra verso destra e da destra verso sinistra ad intervalli di tempo alternati (Fig. 4). I processori, in cui i tre flussi di dati si incontrano, computano il relativo prodotto interno $R_V:=R_y+R_A \cdot R_x$.

Il numero di processori per effettuare tale prodotto è pari all'ampiezza di banda della matrice, w=n+n–1, in quanto la rotazione a 45° impone che gli elementi dell'antidiagonale vengano memorizzati a distanza d=2 lungo un asse parallelo alla catena di processori.

Il numero di passi di computazione è 2n+w–1, inclusi i tempi di I/O; sono infatti, richiesti w+1 passi per ottenere la prima componente di y ed altri 2 (n–1) per ottenere tutte le altre, che vengono emesse ad intervalli di tempo pari a due l'una dall'altra.
Più in generale, per matrici di dimensioni mxn sono necessari 2n+w passi con w=n+m–1.

In conclusione, l'algoritmo ha complessità di tempo O(n) e usa un numero di processori che è anch'esso O(n).

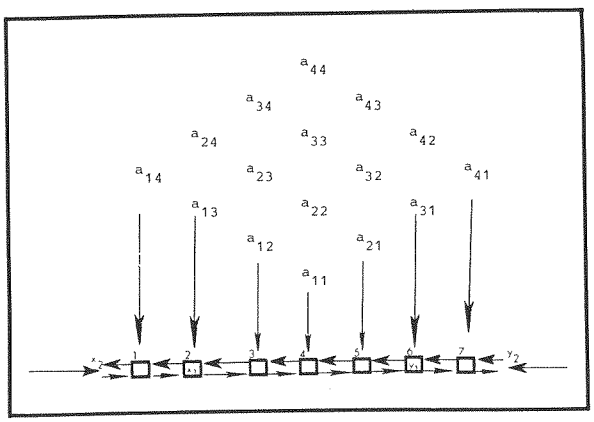

Fig. 4 - Processori linearmente connessi per il prodotto fra una matrice A (4x4) ed un vettore X (4).

5.2 Prodotto fra Matrici

Date due matrici quadrate di ordine n, A e B, si voglia calcolare C=AxB. Si consideri, allora, un insieme di processori esagonalmente connessi. Ogni matrice, ruotata di 45° e memorizzata in w=n+n–1 unità di memoria, lascia entrare lungo le diagonali del reticolo i suoi elementi ad intervalli di tempo regolari. Nei processori in cui gli elementi delle tre matrici A, B e C si incontrano, viene computato il prodotto interno $R_C:=R_C+R_A \cdot R_B$ (Fig. 5).

Il tempo impiegato è 5n-3, compresi i tempi di I/O: infatti, n-1 passi sono necessari affinchè abbia luogo il primo prodotto interno al centro del reticolo; n passi sono necessari per ottenere la prima componente di C; (2n-1) -1 passi sono richiesti dalla computazione di tutte le altre componenti ed, infine, altri n passi sono necessari per avere in output $C_{n,n}$.
Il numero di processori è $3n^2–3n+1$ ed è pari al numero di elementi da elaborare meno il numero di elementi che vengono messi in comune.

Per matrici nxm il numero di processori nel reticolo è sempre $O(n^2)$ ed in particolare è uguale al prodotto $w_1 \cdot w_2$, dove w_1 e w_2 sono le ampiezze di banda delle matrici.
Nel caso che si debbano eseguire prodotti tali che il numero di processori richiesti sia maggiore di quello disponibile, per matrici tridiagonali, indipendentemente dalle loro dimensioni, si può utilizzare sempre lo stesso reticolo costituito da $w_1 \cdot w_2$ processori. Al

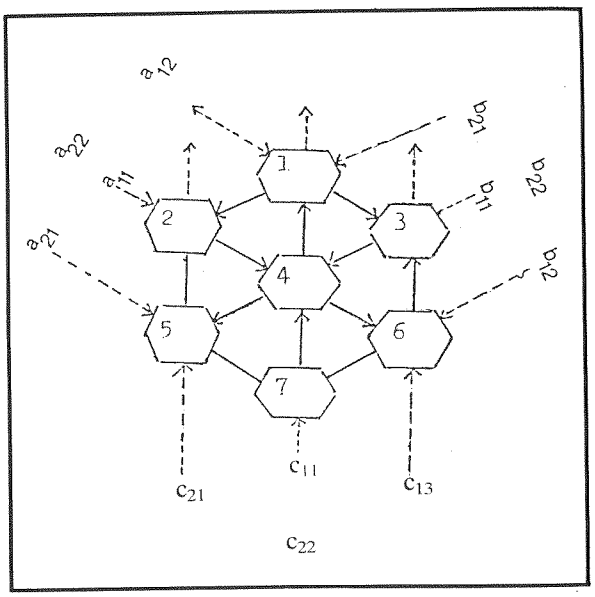

Fig. 5 - Processori esagonali connessi per il prodotto fra due matrici quadrate di ordine due.

contrario se si dovessero, prima, trasformare le matrici in matrici tridiagonali mediante un algoritmo seriale, l'algoritmo complessivo diventerebbe molto inefficiente. Per ovviare a questo inconveniente si può decomporre il prodotto in n^3/K^3 sottoprodotti fra matrici di ordine K, dove n è la dimensione delle matrici assegnate e K la massima dimensione trattabile con il reticolo a disposizione. Ciascun prodotto richiede, allora, (5K-3) passi, cosicchè il tempo complessivo risulta pari a (5K-3) (N^3/K^3).

6. BIBLIOGRAFIA

[1] Kung, H.T., e C.E. Leiserson, SYSTOLIC ARRAYS FOR (VLSI), in "Introduction to VLSI systems" by C.A. Mead, L.A. Conway, Sect. 8.3, Addison-Wesley, Reading, Mass, 1978.

[2] Foster, M.J. and H.T. Kung, THE DESIGN OF SPECIAL PURPOSE VLSI CHIPS, Res. Computer, 13, n. 1, 1980, 26-40.

[3] Guibas, L.J., H.T. Kung e C.D. Thompson, DIRECT VLSI IMPLEMENTATION OF COMBINATORIAL ALGORITHMS, Proc. Caltech Conf. VLSI, 1979, pp. 509-525.

[4] Kuck, D.J., MULTIOPERATION MACHINE COMPUTATIONAL COMPLEXITY, Proc. of Symp. on Complexity of Sequential and Parallel Numerical Algorithms, J.F. Traub, Ed. Academic Press, New York, N.Y. 1973, pp. 17-48.

[5] Thompson, C.D., AREA-TIME COMPLEXITY FOR VLSI, Proc. 11th Ann. A.C.M. Symp. Theory Comput., IEEE, New York, 1979.

[6] Traub, J.F., COMPLEXITY OF SEQUENTIAL AND PARALLEL NUMERICAL ALGORITHMS, Academic Press, New York, N.Y., 1973.

[7] Stone, H.S., PROBLEMS OF PARALLEL COMPUTATION, Proc. of Symp. on "Complexity of Sequential and Parallel Numerical Algorithms", J.F. Traub, Ed. Academic Press, New York, N.Y., 1973, pp. 1-14.

[8] Budnik, P., D. Kuck, THE ORGANIZATION AND USE OF PARALLEL MEMORIES, IEEE Trans. Comp., C-20, 1971.

[9] Burnett, G.J., E.G. Coffman, A STUDY OF INTERLEAVED MEMORY SYSTEMS, Proc. AFIPS 1970 Spring Toint Comp. Conf., vol. 36.

[10] Lawrie, D.H., ACCESS AND ALIGNMENT OF DATA IN AN ARRAY PROCESSOR, IEEE Trans. on Comput., vol. C-24, n. 12, 1975, pp. 1145-1155.

[11] Stone, H.S., PARALLEL PROCESSING WITH THE PERFECT SHUFFLE, IEEE Trans. on Comput., vol. C-20, n. 2, 1971, pp. 153-161.

[12] Kung, H.T., THE STRUCTURE OF PARALLEL ALGORITHMS, Advances in Computers, vol. 19, 1980, pp. 65-108.

[13] Kung, H.T. LET'S DESIGN ALGORITHMS FOR VLSI SYSTEMS, Proc. Caltech Conf. "VLSI: Architecture, Design, Fabrication", Institute of Technology, Pasadena, California, 1979.

[14] Flynn, M.J., VERY HIGH-SPEED COMPUTING SYSTEMS, Proc. IEEE 54 (12), 1966, pp. 1901-1909

H - UN LINGUAGGIO DI PROGRAMMAZIONE LOGICA

Roberto Ghislanzoni, Vieri Samek Ludovici, Giorgio Tornielli
Istituto di Cibernetica - Università di Milano

RIASSUNTO

In questo articolo viene presentato H, un linguaggio di programmazione basato sulla logica del primo ordine. Viene dato un inquadramento generale sull'evoluzione dei linguaggi di programmazione, e viene in particolare trattata la famiglia dei linguaggi logici, al cui interno si colloca H, per evidenziarne differenze e somiglianze con i linguaggi consimili.
Viene infine presentato in dettaglio l'interprete del linguaggio, supportando con esempi i punti ove più la comprensione può risultare difficoltosa.

ABSTRACT

Our aim is to give the reader an overview of our language, and to point out its relationship with other languages, especially those for which logic (in a wide sense) is a major concern, i.e. PROLOG.
The article also present the general algorithm on which H is based, and gives the reader some examples that can help to understanding the contents.

1. DALLA LOGICA AD H

Nel corso degli ultimi decenni, è cresciuto sensibilmente il numero dei linguaggi di programmazione disponibili, ognuno con caratteristiche a volte simili, a volte dissimili, dagli altri: tutti hanno in comune un unico punto fondamentale, rappresentato dal fatto di essere basati sulla manipolazione di simboli.
Per i linguaggi più conosciuti, quali FORTRAN, COBOL, PASCAL, etc., vanno esplicitate operazioni elementari, quali l'assegnamento, che hanno una corrispondenza diretta con il comportamento della macchina, con l'effetto di negare al programmatore la possibilità di trattare simboli a un livello più generale.

Sul finire degli anni '50 è stato disegnato e implementato un linguaggio con lo scopo di fornire strumenti più adatti per la manipolazione simbolica in senso più lato: era nato così il LISP (LIST Processing) [1], un linguaggio fondato esclusivamente sulle associazioni di simboli e sulla loro trasformazione mediante funzioni.

In un programma LISP troviamo differenze concettuali notevoli con i linguaggi tradizionali, la più importante delle quali è l'uniformità di rappresentazione tra dati e codice: il linguaggio LISP vede tutto sotto forma di liste, senza distinzione fra i dati richiesti dal programma o il programma stesso.
Questo porta ad un livello di astrazione più alto rispetto ai linguaggi tradizionali, ed è quindi possibile scrivere un programma che "lavora" con il programma stesso, visto, a seconda delle necessità, o come dato da utilizzare o come utilizzatore: per esempio, in LISP è possibile definire funzioni costruttrici di funzioni, queste ultime manipolate come dato in una parte dell'elaborazione, e poi utilizzate come funzioni vere e proprie in una seconda parte.

Sempre sotto la spinta dell'esigenza di lavorare ad un livello di astrazione più alto, sono nati anche linguaggi più strettamente connessi con il formalismo della logica del primo ordine e quindi con la manipolazione delle proposizioni e del valore di verità ad esse associato: PROLOG [2] è l'esempio più noto, di questi.

Un programma logico è una sequenza di asserzioni, condizionate e non, che costituiscono una base dati in cui "versiamo" la nostra conoscenza su un particolare ambito. Queste conoscenze riguardano le sole proprietà logiche e le relazioni presenti nell'ambito studiato, non il loro ordine di manipolazione. Quest'ultimo aspetto algoritmico è demandato all'interprete.

Un semplice esempio per chiarire questo concetto è il seguente: supponiamo di dover ordinare secondo un ordine crescente alcuni numeri, utilizzando il metodo di ordinamento QUICKSORT [3]: per risolvere questo problema con un linguaggio tradizionale, quale Pascal, dobbiamo innanzitutto presentare i dati al programma sotto una ben precisa forma, per lui comprensibile e soprattutto, dobbiamo specificare rigidamente la sequenza di esecuzione che intendiamo seguire: vale a dire somma questo, poi sottrai, moltiplica etc.

Se dovessimo affrontare il medesimo problema in PROLOG o H ci sarebbe sufficiente introdurre nella base dati la nostra conoscenza sulle operazioni come una procedura che spezza una lista di numeri in due parti, le ordina separatamente (chiamando sè stessa) e infine, la unisce fra di loro producendo la lista ordinata (vedi esempi).

Un programma per la soluzione di un problema complesso però non consta solo di una base dati e della conoscenza racchiusa in essa, ma è anche identificato dalle operazioni che intendiamo svolgere nell'ambito considerato: pertanto è opportuno poter disporre di un linguaggio che, quando serve conoscenza sul problema, si comporti come PROLOG, e invece, quando è necessaria conoscenza più "algoritmica", sia in grado di specificarla. Sulla falsariga di queste considerazioni, è nato H: da un'idea originale di Chester [4], riportata da Simmons [5], è stato implementato presso il Silab dell'Istituto di Cibernetica dell'Università degli Studi di Milano, un interprete per un linguaggio logico, maggiormente integrato però con l'ambiente d programmazione LISP, linguaggio utilizzato per la stesura dell'interprete.

Sostanzialmente, si può dire che H realizza la filosofia del "se... allora", rispecchiando quella logica di cui anche l'uomo a volte fa uso: per questo la programmazione in H, una volta superate le barriere per la comprensione della sintassi, dovrebbe, per così dire, risultare quasi "naturale".

Ma vediamo l'interprete H più in dettaglio.

2. STRUTTURA DEI PROGRAMMI

Un programma H è una lista di clausole alla Horn: si tratta di una clausola composta da una disgiunzione di formule atomiche di cui al più una è positiva (non negata). Una formula atomica consta di un predicato e un numero qualsiasi di argomenti. "(SPOSI Franca Franco)" è un formula atomica: il predicato è SPOSI, argomenti e significato della formula sono immediati. Una formula atomica meno immediata è "(STUDENTE x)" dove x è un argomento variabile. "x" può assumere il nome di qualsiasi persona e la formula affermerà che quella persona è uno studente.
Tornando alle clausole alla Horn: se A1...An e B sono n+1 formule atomiche allora

(non A1) V (non A2) V .. V (non An) V B

(il simbolo "V" si legge "oppure")
è una clausola alla Horn dove B è l'eventuale formula atomica non negata.
Con una trasformazione notazionale possiamo riscrivere la stessa clausola come

B ⟵ A1 & A2 & .. & An

che si legge "A1 e A2 e.. e An implicano "B") senza alterare le sue proprietà logiche.
Se A1 .. An e B sono formule atomiche a cui diamo un significato, come per "(SPOSI Franca Franco)", viene naturale associare alla nuova formula un significato del tipo "quando A1 e .. An sono vere tutte assieme è vera anche "B".
Quest'ultima lettura sembra identica alla precedente ma in realtà mette in luce come dimostrare la verità di B si riduca a verificare la veridicità delle A1 .. An, un problema, che deve essere rappresentato mediante una formula atomica B (conseguente) e che può essere composto in sottoproblemi A1 .. An, viene facilmente descritto con una clausola.
Ad esempio, per sapere se x ha intenzione di rubare l'oggetto y, si deve sapere che x è un ladro e che è attratto dall'oggetto y; in H scriveremo:

(RUBA x y) ⟵ (È_LADRO x) (È_ATTRATTO_DA x y)

Il processo di riduzione a sottoproblemi si può ripetere per ogni formula delle A .. An scrivendo nuove clausole alla Horn contenenti le riduzioni ad ulteriori sottoproblemi più semplici.
Così si potrebbe anche dire:

(È_ATTRATTO_DA x y) ⟵ (È_DI_VALORE y)

Uno stesso problema può ridursi a sottoproblemi in diversi modi: in questo caso scriveremo più clausole distinte che avranno in comune il problema che viene scomposto e più insiemi diversi di sottoproblemi, come in queste clausole

(MADRE_LEGALE x y) ⟵ (FIGLIO y x)
(MADRE_LEGALE x y) ⟵ (FIGLIO_ADOTTIVO y x)

in cui abbiamo scomposto il problema di sapere se x è legalmente la madre di y, una prima volta riducendolo al problema di sapere se è il figlio di x, poi nuovamente riducendolo al problema di sapere se y è un eventuale figlio adottivo di x.
Le clausole alla Horn consentono di calare nel programma anche delle conoscenze di base alle quali ridurre i vari problemi.
Infatti tornando alla definizione, se A1 .. An sono assenti, rimane solo la formula atomica B e nella nostra interpretazione questo significa che B non si riduce a nessun problema ed dunque B è sempre vero.
Così se sappiamo che Marco è figlio di Marta basterà scrivere la clausola:

(FIGLIO Marco Marta)

queste clausole vengono anche chiamate "fatti". Dei fatti sono anche le formule:

(È_LADRO Arsenio_Lupin)
(È_DI_VALORE diamante)

Un programma H consiste solo di queste informazioni: varie clausole che determinano quali problemi possono essere ridotti a quali altri, e altre clausole che incorporano la conoscenza di base a cui arrivar man mano che si riducono i problemi in sottoproblemi.

Come si vede non è assolutamente specificato come queste conoscenze vanno usate, quali usare prima e quali dopo, gli unici aspetti specificati sono quelli logici: una base di conoscenze e come passare da un problema alle varie possibili scomposizioni di esso (diciamo subito che questo non è del tutto vero e che parleremo di un artificio che permette di influenzare l'ordine di computazione, ma lo spirito alla base di H è quello descritto).

Ci limitiamo a fornire all'interprete i problemi (uno o più) che vogliamo risolvere sotto forma di formule atomiche. L'interprete applicherà le regole di riduzione e la conoscenza interna al programma sulle formule atomiche che gli abbiamo fornito nel tentativo di ridurle ad un insieme di "fatti".
L'effetto della riduzione è quello di calcolare il valore delle eventuali variabili presenti nella/e formula/e problema: al programma scritto in precedenza avremmo potuto chiedere che oggetto potrebbe essere rubato da Arsenio Lupin, fornendo il problema

(RUBA Arsenio_Lupin x)

la risposta sarebbe stata

(RUBA Arsenio_Lupin diamante)

Questo modo di eseguire un programma caratterizza i programmi in H rispetto a quelli scritti con linguaggi più tradizionali. In Pascal e altri linguaggi simili va specificato l'inizio del programma scrivendo la parola speciale "PROGRAM" cui segue il nome del programma. L'esecuzione del programma partirà sempre da un punto specifico. In H non c'è un inizio del programma specificato, in un certo senso non c'è un programma. Un programma H è, come già detto, una lista di clausole alla Horn. Qualsiasi regola applicabile alla formula problema è un possibile inizio del programma, ma è meglio pensare a regole di riduzione applicabili di volta in volta ad una formula, piuttosto che ad un programma con un inizio ed una fine.

3. FUNZIONAMENTO DELL'INTERPRETE

L'algoritmo centrale dell'interprete risolve un problema descritto come formula atomica, riducendola in un insieme di formule sottoproblema in accordo con qualche clausola presente nel programma; il processo di riduzione viene ripetuto su ogni sottoproblema accumulato, fino a che le uniche clausole di riduzione applicabili sono dei "fatti". In questo processo di riduzione a sottoproblemi ogni volta va deciso quale clausola applicare.
Applicare una clausola non è sempre possibile: la selezione delle clausole applicabili è fatta mediante un processo detto di "UNIFICAZIONE".
Innanzitutto la formula da ridurre e il conseguente della clausola devono avere lo stesso numero di argomenti.
Successivamente si confrontano gli argomenti della formula da ridurre con quelli del conseguente ad uno ad uno: il primo della prima formula con il primo del conseguente, il secondo della formula con il secondo del conseguente e così via.
Ad esempio dovendo ridurre la formula:

(MEDICO Rossi)

con la regola:

(MEDICO x) ⟵ (ISCRITTO_ALBO_MEDICI x)

la variabile x viene legata al valore Rossi e la verità della prima formula viene ridotta alla verifica dell'unico sottoproblema:

(ISCRITTO_ALBO_MEDICI Derosa)

Ma l'unificazione non è in un solo senso, infatti avendo una base dati del tipo:

(IMPIEGATO Rossi Fiat)
(IMPIEGATO Bianchi Ferrari)
. . .

dall'ovvio significato, possiamo sfruttare l'unificazione in tre modi diversi:

1) Chiedendo se una persona a noi nota è impiegata in una azienda anch'essa nota, come in:

(IMPIEGATO Bianchi Ferrari)

Qui non ci sono variabili in nessuna della due formule da unificare e dunque è sufficiente controllare l'identità fra gli argomenti dell'una e dell'altra formula. Nel nostro caso avremo unificazione solo con il secondo "fatto" della base dati.

2) Chiedendo chi è implicato e dove, come in:

(IMPIEGATO x y)

Qui ogni fatto della base dati unifica con la formula, assegnando ogni volta ad x il nome dell'impiegato ed

a y quello dell'azienda riportando quindi più soluzioni.

3) Chiedendo chi lavora in una azienda particolare, come in:

(IMPIEGATO x Fiat)

Qui potranno unificare tutti e soli quei fatti della base dati che hanno la costante "Fiat" come secondo argomento. In x verranno riportati man mano tutti gli impiegati che lavorano alla Fiat.

Per ultimo una curiosità utile, la formula:

(IMPIEGATO x x)

che ha la stessa variabile come primo e secondo argomento può unificare solo se nella base dati esiste un fatto con la stessa costante come primo e secondo argomento, cioè un fatto del tipo:

(IMPIEGATO Rossi Rossi)

Le soluzioni di volta in volta riportate in x ci espliciteranno dunque tutte quelle persone che lavorano in un'azienda che porta il loro stesso nome.

L'interprete si avvale di due funzioni, la PROVE e la MATCH: la PROVE serve a dirigere il processo di scomposizione in sottoproblemi, la MATCH a confrontare le formule atomiche come abbiamo descritto sopra.

PROVE gestisce uno stack dei problemi da risolvere. Alla base dello stack è la formula "F" che vogliamo dimostrare. PROVE seleziona le clausole con lo stesso predicato di "F" e chiama MATCH a confrontarle una per volta con "F".
Se nel programma non esistono formule unificabili con "F", la dimostrazione è fallita. Se invece ne esiste una, questa può essere o un fatto o una regola di riduzione a sottoproblemi. Nel primo caso abbiamo determinato la dimostrazione in quanto abbiamo trovato un fatto che dimostra "F": PROVE toglie l'elemento in cima allo stack, (cioè "F"), e passa a risolvere il nuovo elemento in cima allo stack.

Nel secondo caso va applicata la regola di riduzione in sottoproblemi e solo se riusciremo a risolvere ognuno di essi sarà dimostrata anche "F".
Allo stack vengono aggiunti tutti i sottoproblemi, ed essendo ognuno un problema da dimostrare, tutto il processo appena descritto si riapplica nuovamente a partire dalla formula in cima allo stack. Ogni formula dimostrata viene estratta dallo stack per passare a dimostrare quella sottostante.

In realtà associata ad ogni formula dello stack c'è una lista di legami contratti dalle variabili in essa contenute, cui sia PROVE che MATCH devono far riferimento sia per conoscere il valore delle variabili che incontrano sia per assegnare nuovi valori alle variabili che ne siano prive.

Spesso esistono più regole di riduzione applicabili ad una stessa formula da dimostrare, rendendo possibili così più soluzioni. PROVE le trova tutte: infatti dopo ogni regola di riduzione che porta ad una riduzione, PROVE simula un fallimento e cerca un'eventuale nuova regola di riduzione che darà una diversa soluzione.

Per esempio, supponiamo di voler dimostrare la formula "P" avendo tre clausole:

P' ⟵ C1 C2
P" ⟵ B1 B2
P"' ⟵ D1

P', P", P"' sono formule con lo stesso predicato di P eventualmente differenti nel numero e nel genere di argomenti, e vengono mantenute dall'interprete in una unica lista per facilitare le operazioni di accesso.
PROVE chiama MATCH a confrontare P con il conseguente della prima clausola.
Supponiamo che l'unificazione fallisca. Allora MATCH confronta P con la seconda clausola, supponiamo con successo. Allora dallo stack:

((P)())

contenente il solo problema P e una lista dei legami naturalmente vuota si passa allo stack:

((B1 B2) ‹lista dei legami PP"›)((P)())

dove è stato aggiunto un intero elemento costituito dalla lista dei due sottoproblemi da dimostrare e dalla lista dei legami (eventuali) delle variabili stabilitisi durante l'unificazione fra P e P".
PROVE cercherà di dimostrare prima B1 poi, se ha trovato una soluzione, proseguirà con B2. Se la dimostrazione su B1 o su B2 dovesse fallire, anche la formula P rimane indimostrata. In questo caso PROVE fa un passo di BACKTRACKING e cioè toglie l'elemento in cima allo stack e riparte con lo stack iniziale:

((P)())

Nella lista delle clausole con predicato P, un puntatore ricorda il punto raggiunto nella scansione. Dunque non ripartiamo da capo (per poi fallire nuovamente e così via), ma scandiamo la lista a partire dall'ultima formula analizzata. La prima clausola che MATCH tenta di applicare è P". Supponiamo che l'unificazione con P"' abbia successo. PROVE costruisce lo stack:

((D1) ‹lista legami P"'›)((P)())

PROVE si applica al primo elemento. Supponiamo che esista un "fatto" D1' su cui l'unificazione ha successo. Questo produce una lista dei legami fra le variabili di D1 e D1' ma poichè D1' è un "fatto" non vengono aggiunti sottoproblemi. Viene aggiunto allo stack un elemento la cui prima parte è vuota:

(( ) ‹lista legami D1 D1'›)((D1) ‹lista legami P P"'›)((P)())

Quando PROVE si applica a questo stack trova una lista di sottoproblemi da dimostrare vuota. Allora toglie questo elemento dallo stack e integra la lista dei legami dell'elemento precedente con quella di questo elemento. Inoltre poichè la dimostrazione ha avuto successo (abbiamo raggiunto dei "fatti") il primo sottoproblema della nuova "cima" dello stack può essere rimosso.
Nel nostro caso viene rimosso D1, e, poichè era l'unico sottoproblema, la lista rimane vuota.

Lo stack diventa:

(( ) ‹lista legami D1 D1' e P P"'›)((P)())

Anche qui PROVE non ha nulla da dimostrare, nuovamente viene eliminato un elemento e il primo sottoproblema dell'elemento sottostante.
Viene invece riportata la lista integrata dai legami.
Lo stack diventa:

(() ‹lista legami D1 D1' P P"'›)

Questo è l'ultimo elemento dello stack. Una dimostrazione è riuscita e la formula P integrata dai nuovi valori delle variabili viene visualizzata.

In realtà PROVE simula un fallimento e riparte a cercare una dimostrazione.
Poichè la funzione MATCH parte sempre dall'ultima clausola raggiunta le nuove soluzioni sono sempre prodotte dall'applicazione di nuove clausole.

Quando questo comportamento non è desiderato lo si altera inserendo nella lista dei sottoproblemi il simbolo "/". Questo simbolo viene riconosciuto da PROVE e allora le liste di clausole relative alle formule che lo precedono vengono potate. Dunque supponiamo che PROVE nel corso di una dimostrazione abbia formato lo stack:

((A B C / D)(‹lista legami›))((..)(..)) .....

PROVE dimostrerà A poi B e poi C. Nel corso delle dimostrazioni la MATCH ricorda il punto raggiunto scandendo le liste con le clausole concernenti A B e C alla ricerca di una con conseguente adatto.
Chiaramente ognuna tra A B e C può ridursi a un insieme di sottoproblemi che andranno a loro volta dimostrati.

Supponiamo che tutto vada bene e si giunga a dover dimostare "/".
Di per sè "/" viene dimostrato subito perchè esiste un fatto identico "/" predefinito; però "/" comporta l'effetto collaterale di svuotare le liste delle clausole, per A B e C, che ancora non erano state utilizzate dall'algoritmo di unificazione.
Fatto questo, D subirà un normale tentativo di dimostrazione. Se tutto va bene anche D è dimostrata e la prima soluzione è visualizzata. PROVE però dovendo fornire tutte le soluzioni, simula un fallimento e riparte. Quando giunge a dimostrare A B e C non può trovare nuove soluzioni perchè la lista delle clausole è stata tagliata.
Questo non è vero per D, che può ancora utilizzare tutte le clausole non ancora applicate.

4. CONFRONTO FRA PROLOG E H

Nei paragrafi precedenti si è visto che una delle idee che stanno alla base di un linguaggio logico (sia esso PROLOG o H) è quella della scomposizione di un certo problema in sottoproblemi, e che il meccanismo base per il controllo delle dimostrazioni è il simbolo di taglio (il "!" in PROLOG o "/" in H) che impedisce ulteriori ritorni su strade già percorse e quindi di riportare soluzioni alternative.

Ma in alcune applicazioni questi elementi non sono sufficienti, infatti a volte è necessario simulare tecniche di programmazione più vicine ai linguaggi tradizionali come le iterazioni o le selezioni.
In PROLOG il risultato è ottenuto introducendo delle primitive extralogiche che però lo allontanano dall'intenzione di vedere ogni istruzione come un'asserzione del calcolo dei predicati.
Come esempio, consideriamo l'operazione di copia di un file f1 su di un file f2 [7].
In un linguaggio tradizionale, dopo aver aperto i due file, uno in lettura e l'altro in scrittura si potrebbe usare un ciclo in cui si legge un elemento per volta dal file sorgente e lo si copia sul file destinazione, fino ad arrivare al fine file.
In PROLOG si deve scrivere una clausola in cui si sfrutta il meccanismo di backtracking ed alcune primitive extralogiche per simulare il ciclo:

```
copyfile (f1, f2) :- open (f1), open (f2),
                repeat,
                    read (f1, X),
                    (X=EOF, close (f2), !;
                     write (f2, X), fail).
```

Le primitive di cui si è detto sono:

1) repeat : un predicato senza argomenti sempre vero
2) ! : il simbolo di taglio, usato per bloccare il backtracking e terminare l'operazione di copia quando si è raggiunto il fine file.

3) fail : un predicato sempre falso, il cui effetto è di forzare un backtracking fino al repeat.

4) (p ; q) : dove nel nostro caso p è "X=EOF, close (f2), !" mentre q è "write (f2 , X), fail" il cui comportamento è paragonabile a quello
di una istruzione di selezione dei linguaggi procedurali: dimostro q solo se p non è dimostrabile.
Nell'esempio, dimostrare p significa aver raggiunto il fine file, chiuderlo e terminare la copiatura grazie al simbolo di taglio.
Dimostrare q significa scrivere sul file destinazione l'elemento letto e ritornare col fail fino al repeat.

Certamente non è immediato associare un significato puramente logico alla clausola precedente. Questo perchè in essa prevale la necessità di controllare l'uso dei predicati "read", "write" e "close", più che di ridurre l'operazione di copia in operazioni elementari distinte.

Generalizzando si può dire che non tutti i problemi possono essere ridotti con facilità ad una congiunzione di operazioni elementari, ma a volte o esistono delle relazioni di causalità fra i vari sottoproblemi, oppure un problema può essere scomposto in vari modi che si escludono a vicenda, vale a dire ove solo uno di essi può essere utilizzato in una dimostrazione.
Sono proprio queste le situazioni in cui i linguaggi della famiglia del PROLOG mostrano i propri limiti.

Vediamo ora quali possibilità offre alla programmazione l'avere a disposizione contemporaneamente, due formalismi distinti.
L'analisi e la codifica di un problema, possono essere condotti su due fronti: uno in cui prevale una conoscenza "dichiarativa" cioè legata a quali sono le proprietà degli oggetti che si vogliono manipolare, che è più facilmente programmabile in H ed un secondo "procedurale" in cui è più noto il meccanismo necessario per ottenere la soluzione che non le proprietà di cui gode, e quindi più legato al modo di programmare LISP.

Nell'esempio della copia da un file sull'altro prevale la conoscenza procedurale ma nonostante ciò in PROLOG, la si deve esprimere nel formalismo logico, mentre sarebbe più facilmente esprimibile in un'altra forma: nel nostro caso il LISP.

Da quanto detto segue che la differenza fondamentale fra H e PROLOG, è più legata alla metodologia di analisi e scrittura dei programmi che non alle caratteristiche logiche dei due linguaggi.

Sia H che PROLOG usano un algoritmo di risoluzione in cui lo sviluppo delle formule sottoproblema è fatta da sinistra verso destra, e quando si hanno più clausole attivabili la risoluzione viene compiuta seguendo l'ordine in cui le clausole stesse sono state disposte all'interno del programma.
Anche in H è stato introdotto il simbolo di taglio per evitare dei backtacking non voluti
Esiste però una differenza rilevante, anche dal punto di vista teorico, fra H e PROLOG nell'algoritmo di unificazione: in PROLOG non viene fatta una unificazione completa. Così è permesso che (A y (f y)) unifichi con (A x x) legando x ad (f x). Logicamente la circolarità non è ammessa, perchè vorrebbe dire che le due formule possono essere rese uguali se x assume il valore (f x) che, a sua volta, dipende dal valore di x e così via.
In H il controllo della circolarità viene sempre effettuato.
È inibito solo se richiesto esplicitamente.

5. LA FUSIONE FRA H E LISP

Per poter scomporre un programma nella componente H e in quella LISP, è necessario che esse comunichino in modo semplice.
Innanzitutto, i due formalismi condividono la sintassi usuale delle liste, che permette di mantenere in H l'uniformità di rappresentazione fra dati e codici tipica del LISP.
Quindi, un programma H è rappresentato come una lista di clausole e analogamente i termini a cui si riferiscono i predicati possono essere a loro volta delle liste.
Inoltre le richieste di dimostrazione sono fatte con delle funzioni LISP che ritornano come valore la lista degli elementi che verificano il predicato dimostrato.
Perciò, i risultati delle dimostrazioni sono oggetti che possono essere ulteriormente maneggiati da programmi LISP.
Consideriamo, ad esempio, una base dati che contiene alcune asserzioni che esprimono l'area di alcuni paesi di kmq

(area Cina 9561)
(area India 3268)
(area Usa 9363)
(area Urss 22404)

Se si chiede quali sono le aree dei paesi presenti nella base dati si otterrà una lista in cui i risultati sono posti nello stesso ordine in cui sono presenti nella base dati e cioè

(9561 3268 9363 22404)

Se si vuole disporre poi la lista in ordine crescente si può scrivere una funzione LISP in grado di farlo e passargli come argomento la lista precedente

Naturalmente è possibile anche che sia H ad usare i risultati di valutazioni LISP: basta associare ad alcuni dei sottoproblemi all'interno di una clausola una funzione LISP. Quando l'interprete H dovrà dimostrare una tale formula, il controllo verrà passato al LISP che valuterà la funzione: il predicato risulterà dimostrato se la funzione ritornerà un valore di verità vero.
Così nell'ipotetica clausola

(A x y) ‹ ... (diversi x y)

dove per poter dimostrare (A x y) si deve controllare che gli argomenti siano diversi fra loro, si può associare al predicato (diversi x y) la funzione

(lambda (x y) (not (equal x y)))

che valutata produrrà il risultato voluto.

6. LIMITI DI H

Il limite principale del linguaggio deriva dal fatto che attualmente il suo interprete è scritto in LISP e che quindi si ha una bassa velocità di esecuzione essendo a sua volta il LISP interpretato.
Si ha anche lo svantaggio di dover notificare esplicitamente all'interprete H quali predicati hanno associato delle funzioni, mentre sarebbe più ragionevole che fosse l'interprete stesso a riconoscerle e a comportarsi di conseguenza.
Entrambi i problemi verranno rimossi in ulteriori sviluppi in cui si realizzerà un interprete unico per LISP e H dove sarà possibile avere una maggiore omogeneità, come avviene già in alcuni linguaggi simili, per finalità, al nostro come LOGLISP [6].

7. ESEMPI DELL'USO DI H

7.1 Base Dati

Come primo esempio, vediamo come H può essere utilizzato per la gestione di una semplicissima e rudimentale base dati per opere letterarie, in cui per ogni autore vengono mantenute delle informazioni riguardanti le opere e la biografia. (La funzione "axioms" serve a definire gli assiomi che costituiscono il programma):

```
(axioms '(
((attività Shakespeare (drammaturgo poeta)))
((CittaNatale Shakespeare (Stratford on Avon)))
((visse Shakespeare 1564 1616))
((genitori Shakespeare John (Mary Arden)))
((opere Shakespeare ((Romeo e Giulietta) (Le allegri comari di Windsor)...)))

((attività Dante (poeta)))
((CittaNatale Dante Firenze))
((visse Dante 1265 1321))
((opere Dante ((Convivio)) (Vita nuova) (Divina Commedia)...)))
.
.
.
]
```

La base dati può essere interrogata usando la funzione "?" nel seguente modo:
(Ogni atomo preceduto dal "_"viene automaticamente riconosciuto come una variabile)

```
(? (visse Dante_nasce_muore))
```

ottenendo come risposta la stessa formula con le variabili sostituite dai valori estratti dalla base dati:

```
(visse Dante 1265 1321)
```

Sfruttando il modo con cui sono state inserite le informazioni nella base dati, si possono definire alcune regole che permettono una interrogazione più agevole della base dati:

```
(axioms '(
((padre _x _padre) ‹(genitori _x _padre _madre))
((madre _x _madre) ‹(genitori _x _padre _madre))
((nasce _x _nasce) ‹(visse _x _nasce _muore))
((muore _x _muore) ‹(visse _x _nasce _muore))
((scrisse _x _opera) ‹(opere _x _op) (appartiene
    _opera _op))

]
```

Le regole possono essere usate allo stesso modo in cui si usavano i fatti direttamente memorizzati nella base dati:

```
(? (madre Shakespeare _nome))
```

ottenendo come risposta

```
(madre Shakespere (Mary Arden))
```

7.2 Funzioni di Utilità

Nell'ultima regola dell'esempio precedente appare il predicato "appartiene" il cui significato è quello di essere dimostrabile se il suo primo argomento è presente nella lista individuata dal secondo argomento.
In H lo si può scrivere nel seguente modo:

```
(axioms '(
((appartiene _x (_x . _resto)))
((appartiene _x (_primo . _resto)) ‹ (appartiene _x
    _resto))
]
```

Cioè o x è il primo elemento della lista (prima clausola), oppure x "appartiene" alla lista privata del suo primo elemento (seconda clausola).

Altra funzione di utilità è "concatena", che dati due elementi, calcola il risultato della loro concatenazione.

```
(axioms '(
((concatena ( ) _lista _lista))
((concatena (_x . _resto) _lista (_x . _lista1)) ‹
          (concatena _resto _lista _lista1))
]
```

Le prime clausole dice che concatenare una lista vuota con una qualsiasi lista produce la lista stessa; mentre la seconda dice che, se la prima lista non è vuota, il risultato della concatenazione, sarà la lista formata dal primo elemento della prima lista e dalla concatenazione della coda della prima lista con l'altra lista.
Il predicato si può usare nel seguente modo:

```
(? (concatena (a b) (c d) _risultato))
```

ottenendo

```
   (concatena (a b) (c d) (a b c d))
```

oppure anche

```
(? (concatena (1 2) _risultato (1 2 3 4 5))
```

ottenendo

```
(concatena (1 2) (3 4 5) (1 2 3 4 5))
```

Dove la variabile _risultato gioca ruoli diversi a seconda della sua posizione.

7.3 Quicksort

Siamo finalmente pronti per poter scrivere l'algoritmo di ordinamento Quicksoft di cui si era parlato nell'introduzione di questa nota:

```
(axioms '(
((quicksort ( ) ( )))
((quicksort (_primo . _resto) _ListarOrdinata) ‹
          (partiziona _primo _resto _minori _mag-
          giori)
          (quicksort _minori _Ordinati1)
          (quicksort _maggiori _Ordinati2)
          (concatena _Ordinati1 (_primo . _Ordi-
          nati2) _ListaOrdinata)]
```

Il predicato "quicksort" è stato commentato in precedenza.
Il predicato "partiziona", dato un elemento (_"primo") ed una lista (_"resto"), ritorna due liste contenenti rispettivamente l'insieme degli elementi minori e maggiori dell'elemento suddetto.

7.4 Grammatica

Qui sotto è riportato un esempio per il riconoscimento di semplici frasi del linguaggio naturale. Abbiamo una prima serie di regole di produzione direttamente derivate dalla seguente grammatica formale:

```
‹frase› ⟶ ‹frase nominale› ‹frase verbale›
‹frase nominale› ⟶ ‹articolo› ‹frase nominale› |
                   ‹preposizione› ‹frase nominale› |
                   ‹aggettivo› ‹frase nominale› |
                   ‹nome› ‹frase nominale› |
                   ‹nome›
‹frase verbale› ⟶ ‹verbo› ‹frase nominale›
‹articolo› ⟶ la
‹preposizione› ⟶ sulla | di
‹aggettivo› ⟶ parecchi
‹verbo› ⟶ costa | corre
‹nome› ⟶ macchina | Carlo | milioni | pista | Fran-
co
```

Ciò che segue è il programma H corrispondente.
Chiariamo il passaggio dalla grammatica al programma descrivendo la prima regola.
Il predicato "frase" ha due argomenti: il primo, "_x", corrisponde alla frase che deve essere riconosciuta, nel secondo viene costruita la rappresentazione lineare dell'albero sintattico della frase, giustapponendo la parola "FRASE" alle variabili "_fn" e "_fv" risultati parziali restituiti dall'analisi compiuta dai due sottoproblemi, "frasenominale" e "fraseverbale", in cui "frase" si scompone. Le regole corrispondenti a "frasenominale" e a "fraseverbale" hanno un ulteriore argomento "_resto" che mantiene la parte della frase che ancora rimane da analizzare.
Chiaramente questa grammatica non ha nessuna pretesa di completezza.

```
(axioms '(

((frase _x  (FRASE _fn _fv)  )‹
        (frasenominale _x _resto _fn)
        (frasenominale _resto nil _fv))

((frasenominale (_x . _y) _resto (FRASNOME
        (ART _x) _fn))‹
        (articolo _x)
        (frasenominale _y _resto _fn))

((frasenominale (_x . _y) _resto (FRASNOME
        (PREP _x) _fn))‹
        (preposizione _x)
        (frasenominale _y _resto _fn))
```

```
((frasenominale (_x . _y) _resto (FRASNOME
        (AGG _x) _fn))‹
        (aggettivo _x)
        (frasenominale _y _resto _fn))

((frasenominale (_x . _y) _resto (FRASNOME
        (NOME _x) _fn))‹
        (nome _x)
        (frasenominale _y _resto _fn))

((frasenominale (_x . _y) _y (NOME _x))‹
        (nome _x))

((fraseverbale (_x . _y) _resto (FRASVERBO
        (VERBO _x) _fn))‹
        (verbo _x)
        (frasenominale _y _resto _fn))


((articolo la))
((preposizione di))
((preposizione sulla))
((nome pista))
((nome macchina))
((nome Carlo))
((nome Franco))
((nome milioni))
((verbo costa))
((verbo corre))
((aggettivo parecchi))
]
```

Una frase riconoscibile dalla grammatica è:

```
(? (frase (la macchina corre sulla pista) _z)))
```

il programma ritorna la stessa formula completata in _z dall'albero sintattico corrispondente:

```
(frase (la macchina corre sulla pista)
    (FRASE
      (FRASNOME
        (ART la)
          (NOME macchina))
      (FRASVERBO
        (VERBO corre)
          (FRASNOME
            (PREP sulla)
              (NOME pista]
```

analogamente con:

```
(? (frase (la macchina di Carlo costa parecchi milioni) _z))
```

si ottiene:

```
(frase (la macchina di Carlo costa parecchi milioni)
    (FRASE
      (FRASNOME
        (ART la)
          (FRASNOME
            (NOME macchina)
              (FRASNOME
                (PREP di)
                  (NOME Carlo))))
      (FRASVERBO
        (VERBO costa)
          (FRASNOME
            (AGG parecchi)
              (NOME milioni]
```

8. RINGRAZIAMENTI

Si ringraziano il prof. Gianni Degli Antoni per il valido stimolo e i suggerimenti prestati durante la costruzione dell'interprete e il dott. Luca Spampinato per il continuo appoggio nella stesura del codice.

9. BIBLIOGRAFIA

[1] McCarthy et al, LISP 1.5 PROGRAMMER'S MANUAL, MIT press. Boston, 1965

[2] Clocksin W.F. and Mellish C.S., PROGRAMMING IN PROLOG, Springer-Verlag, Berlin, 1981

[3] Wirth N. ALGORITHMS + DATA STRUCTURES = PROGRAMS, Prentice-Hall, Englewood Cliffs, New Jersey, 1976

[4] Chester D., USING HCPRVR, Department of Computer Science, University of Texas, Austin, June1980

[5] Chester D., HCPRVR: A LOGIC PROGRAM INTERPRETER IN LISP, proc. AAAI, Dept. Comp. Sc., University of Texas, Austin, 1980.

[6] Simmons R.F., COMPUTATIONS FROM THE ENGLISH, Prentice-Hall, New Jersey, 1984

[7] Robinson J.A. and Silbert E.E., LOGLISP - AN ALTERNATIVE TO PROLOG, Comp. Sc., Syracuse University, Syracuse, 1980

[8] Mc Dermott D., THE PROLOG PHENOMENON, SIGART newsletter, 1980

APPENDICE

```
(setq limit 200        ;limite di profondità
                       ;ammesso per lo
                       ;sviluppo dell'albero
      shortform t      ;quando messo
                       ;al valore t
                       ;non controlla se
                       ;‹x› è legato a ‹f x›
      l_trace nil      ;inibisce il tracciamento delle funzioni e dei
                       ;predicati quando messo a "nil"
      int_mode 'all) ;Valore di default per ottenere tutti i valori
                       ;delle variabili che verificano i predicati che si
                       ;vogliono dimostrare.

(setq functions nil    ;contiene tutte le funzioni definite con "attach"
      variables nil    ;contiene tutte le variabili usate
      predicates nil   ;contiene i predicati che stanno alla sinistra
                       ;delle regole e definiti mediante "axioms"
lisp_use ( ))          ;variabile usata quando si usano le funzioni
                       ;"all", "the" e "any"


; le seguenti funzioni sono le funzioni usate per richiedere delle di-
; mostrazioni all'interprete. L'uso della funzione "?" è stato descritto
; negli esempi.
; le funzioni "the" e "all" ritornano come valore rispettivamente il
; primo valore o tutti i valori che verificano la dimostrazione.
; Con "any" si può anche specificare quanti valori si vogliono cal-
; colare.
; A differenza di "?", queste ultime tre funzioni permettono di
; specificare quali sono le variabili di cui si vuole calcolare il valore.
; Così per calcolare tutti gli _x che verifichino il predicato A, si può
; chiedere (all '_x '((A _x)))
; che ritornerà la lista dei valori cercati.

(def ?
  (nlambda (formulas)
    (mkvars formulas)
    (*catch 'first_level
      (let ((count (1 + limit))
            (globalslashflag nil))
           (prove (list (list formulas nil)) 1)))

    nil]

(defun all (formulas cong)
       (mkvars formulas)
       (let ((globalslashflag ( ))
             (int_mode 'all)
             (lisp_use t)
             (count (1 + limit))
             (lisp_channel ( )))
         (prove '((,cong ( )) 1)
         lisp_channel]

(defun the (formulas cong)
       (mkvars formulas)
       (car (*catch 'first_level
              (let ((globalslashflag ( ))
                    (int_mode 'first)
                    (lisp_use t)
                    (count (1 + limit))
                    (lisp_channel ( )))
                    (prove '((,cong ( )) 1)]

(defun any (quant formulas cong)
     (mkvars formulas)
     (*catch 'first_level
       (let ((globalslashflag ( ))
             (int_mode (1-quant)
             (lisp_use t)
             (count (1+limit))
             (lisp_channel ( )))
             (prove '((,cong ( )) 1)]
```

```
; la funzione "match" realizza l'unificazione delle formule "x" con
; la formula "y". Le variabili "xlist" e "ylist" rappresentano gli
; ambienti da cui si possono estrarre i valori relativi alle variabili
; presenti nelle due formule.
; il valore ritornato dalla funzione è "t" nel caso le due formule sia-
; no unificabili, "nil" nel caso non lo siano.
; Come effetto collaterale aggiorna lo stack con i nuovi valori delle
; variabili ottenuti durante il processo di unificazione.

(defun match (x xlst y ylst)
(prog ( )
a (cond ((atom x)
  (cond ((not (numbp x))
    (cond ((get x 'variable)
       (cond ((setq temp ((setq temp (assoc x (car xlst)))
              (setq x (cadr temp)
              (setq xlst (cddr temp))
              (go a))
              (t (prog ( )
                b (cond ((and (atom y)
                              (not (numbp y))
                              (get y 'variable)
                              (setq temp (assoc y (car ylst))))
                         (setq y (cadr temp))
                         (setq ylst (cddr temp))
                         (go b))))
                  (cond ((and (eq x y) (eq xlst ylst))
                         (return t))
                        ((or shortform (absent x xlst y ylst))
                         (setq save (cons xlst save))
                         (rplaca xlst
                              (cons (cons x (cons y ylst))
                                    (car xlst)))
                         (return t))
                         (t (return nil))))))


             (t (go c))))
       (t (go c)))))
b (cond ((atom y)
        (cond ((not (numbp y))
              (cond ((get y 'variable)
                     (cond ((setq temp (assoc y (car ylst)))
                            (setq y (cadr temp)
                            (setq ylst (cddr temp))
                            (go b))
                            (t (cond ((or shortform (absenty
                                                    y ylst x xlst))
                                      (t (cond ((setq save (cons ylst save))
                                                (rplaca ylst
                                                  (cons (cons y (cons x
                                                                xlst))
                                                        (car ylst)))
                                                (return t))
                                                (t (return nil))))))
                            (t (return (eq x y)))))
                    (t (return (eq x y))))))
(cond ((match (car x) xlst (car y) ylst)
       setq x (cdr x))
       setq y (cdr y))
       go a))
       (t (return nil)))

c (cond ((atom y)
        (cond ((not (numbp y))
              (cond ((get y 'variable)
                     (cond ((setq temp (assoc y (car ylst)))
                            (setq y (cadr temp)
                            (setq ylst (cddr temp))
                            (go c))
                            (t (cond ((or shortform (absent
                                                    y ylst x xlst))
                                      (t (cond ((setq save (cons ylst save))
                                                (rplaca ylst
                                                  (cons (cons y (cons x
```

```
                                                        xlst))
                                                (car ylst)))
                                        (return t))
                                        (t (return nil))))))
                                (t (return (eq x y)))))
                        (t (return (eq x y))))))

; La funzione "absent" realizza il controllo di occorrenza, cioè
; controlla se una variabile è legata ad un termine che la contiene,
; oppure no.

(defunt absent (x xlst y ylst)
  (prog ( )
    a (cond ((atom y)
             (cond ((or (numbp y) (not (get y 'variable)))(return t))
             ((setq temp (assoc y (car ylst)))
              (setq y (cadr temp))
              (setq ylst (cddr temp))
              (go a))
             ((return (not (and (eq xy) (eq xlst ylst)))))))
             ((absent x xlst (car y) ylst)
              (setq y (cdr y))
              (go a)))))

; la funzione "index" estrae il valore dell'atomo "z" dall'ambiente
; "zlst"
; è usata per permettere di avere dei predicati "variabili"

(defun index (z zlst)
  (prog ( )
    a (cond ((not (atom z)) (setq z (car z))(go a))
            ((setq temp (assoc z (car zlst)))
             (setq z (cadr temp))
             (setq zlst (cddr temp))
             (go a))
             (z (return z)))))

; la funzione "prove" ha il compito di mantenere lo stack necessario
; nello sviluppo depth first dell'albero di risoluzione.

(defun prove (stack level
  (prog (save ASS z w)
    (setq count (1- count))
  c (cond ((zerop count)
           (princ "continue")
           (cond ((member (setq command (read)) '(y yes))
           (cond ((setc    unt (1+ limit)))
yes))
          ((member command '(n no)) (*throw 'first_level))
           (t (eval command))(go c)))))
b (cond ((null (caar stack))
         (setq level (1- level))
         (cond ((null (cdr stack))
                (setq val (expand1 (cdar stack) formulas))
                (cond (lisp_use
                     (setq lisp_channel (cons (expand1 (cdar stack)
                                                       formulas)
                                              lisp_chennel))
                      (t (write val)))
                (cond ((another?)
                       (return nil)); simulo un fallimento
                       (t (*throw 'firt_level lisp_channel))));
termino l'esecuzione
               ((setq stack (cdr stack)
                (or (and l_trace
                    (write level "<-" (expand1 (cdar stack)
                                               (caaar stack))))
                    t)
                (setq stack (cons (cdaar stack) (cdar stack))
                                                  (cdr stack)))
                (go b))))
        ((isfunc? (expand 1 (cdar stack) (caaar stack)))
         (cond (l_trace (write level "f ->" (expand1 (cdar stack)
                                                (caaar stack)))))
         (cond ((apply (get (expand1 (cdar stack)
                                    (car (caaar stack))) 'funct)
                       (expand1 (cdar stack) (cdr (caaar stack))))
                (setq stack (cons (cons (cdaar stack) (cdar stack))
                                                  (cdr stack)))
                go b)))))
(setq ASS (get (index (caaar stack) (cdar stack)) 'axiom))
a (cond ((zerop count) (return))
        ((null ASS)
         (cond ((eq (caaar stack) '/)
                (or globalslashflag (setq globalslashflag
                                                  (cdar stack)))))
         (return))
        ((and (match (caar ASS)
                     (setq z (list nil)))
                     (caaar stack)
                     (cdar stack))
              (or (and l_trace
                       (write level "->" (caaar stack))
                       (write "      "(expand1 z (car ASS))))
         t)
             (setq w (prove (cons (cons (strip (cdar ASS)) z) stack)
                                                  (+ level 1))))
         (return w))
        (t (map_null sav    'mbda (x) (rplaca x (cdar x))))
          (setq save nil)
          (setq ASS (cond ((null globalslashflag) (cdr ASS))
                          ((eq globalslashflag z) (setq
                                                globalslashflag ( )))))
          go a)))))

(def another? (lambda ( )
  (cond ((eq int_mode 'all) t)
        ((or (eq int_mode 'first)
             zerop int_mode))
          nil)
        ((numbp int_mode)

         (setq int_mode (1- int_model))
         t)
        (t (continue?]

(def continue? (macro ( )
  '(progn (princ "another")
          (member (read) '(y yes))))]

(def isfunc? (macro (name)
  `cond ((atom ,(cadr name( ) nil)
         t (get (car, (cadr name)) 'funct))))]

(def strip (macro (x)
  '(cond ((null ,(cadr x)) ( ))
  `cond (t (cdr ,(cadr x)))))]

; La funzione "expand1" ha come argomenti un ambiente e una
; espressione, e ritorna l'espressione stessa ove, le eventual variabili
; presenti sono state sostituite dal loro valore estratto dall'ambiente
; stesso.

(defun expand1 (lst x)
  (cond ((not (atom x)) (cons (expand1 lst (car x)) (expand1 lst
                                                  (cdr x))))
        ((numbp x) x)
         ((get x 'variable)
         (cond ((setq temp (assoc x (car lst)))
                (expand1 (cddr temp) (cadr temp)))
               (t x)))
        (t x)))

; le seguenti sono le funzioni usate per gestire l'inserimento e la
; cancellazione di formule dalla base di dati.
; Fra di esse ci sono anche delle funzioni necessarie per l'espansione
; di alcune forme speciali come "if", "with" e "or" che non sono
; però state descritte nell'articolo.

(defun axioms (axioms_list)
```

```
(*catch 'first_level
   (let ((assert_numb Ø))
        (map_null axioms_list 'rmAxiom)
        (map_null axioms_list 'build_assert))))

(defun axioms (axioms_list)
  (*catch 'first_level
     (let ((assert_numb Ø))
          (map_null axioms_list 'built assert))))

(defun rmAxiom (assert)
          (and (listp assert)
          (putprop (predicate_of assert) ( ) 'axioms))]

(defun mkvars (assert)
  (cond ((null assert) nil)
          ((not (atom assert)) (cons (mkvars (car assert)) (mkvars
                                                  (cdr assert))))
          ((eq (car (explode assert)) '_)
           (cond ((not (member assert variables)
                  (putprop assert t 'variable)
                  (setq variables (cons assert variables))))
           assert)
          (t assert)))

(def build_var (macro (body)
  '(implode (cons '? (cdr (explode, (cadr body))))))]

(defun build_assert (x)
       (setq assert_numb (1+ assert_numb))
       (macro_axiom (mkvars x)))

(defun macro_axiom (assert)
  (cond ((fact? assert) (new_axiom assert))
          ((with? assert) (build_with assert))
          ((not_correct assert)
           (write "Manca il segno di implicazione in "assert_numb)
                                          (*throw 'first_level))
          ((primo_ante assert) 'if) (built_if assert))
          ((primo_ante assert) 'or) (built_if assert))
          (t (new axiom assert))))

(def fact? (macro (body) `(null (cdr ,(cadr body))))]
(def primo_ante (macro (body) '(caddr ,(cadr body)))]
(def not_correct (macro (body) '(not (eq (cadr ,(cadr body)) '‹)))]

(defun build_with (assert)
     ;assert = (with pred_name ×(istanza)⇔)
     (let ((pred_name (cadr assert)) (istanze (cddr assert)))
          (until (null istanze)
                 (cond ((eq (caar istanze) '‹)
                        (write "Uso improprio del costrutto with in
                                                  "assert_numb)
                        (*throw 'first_level))
                       (t (new_axiom '((,pred_name,    (nextl‹
                                                  istanze)))))))))

(def with? (macro (body) '(eq (car, (cadr body)) 'with)))



(defun build_if (assert)
  (let ((conseg (car assert)) (discr (cadddr assert))
       (ramo_then (cadddr assert)) (ramo_else (caddddr assert)))
       (cond ((eq (car discr) 'and)
              (macro_axiom '(,conseq ‹,    (cdr discr) / ,
                                                  ramo_then))
             ((eq (car discr) 'or)
              (nextl discr)
              (until (null discr)
                     (macro_axiom '(, conseq ‹ ,(nextl discr) / ,
                                                  ramo_then))))
             (t (macro_axiom '(conseq ‹ ,discr / ,  ramo_then))))
       (cond ((not (null ramo_else))
              (macro_axiom '(,conseq ‹ ,  ramo_else))))))
```

```
(defun build_or (assert)
 ;assert = (conseq ‹ or ×(A1...An)±)
 (let ((conseq (car assert)) (altern (cdddr assert)))
      (cond ((null altern))
            (write "Struttura or scorretta in "assert_numb)
            (*throw 'first_level))
           (t (until (null altern)
                     (macro_axiom '(,conseq ‹ ,   (nextl altern))))))

(defun new_axiom (assert)
  (cond ((or (atom assert) (atom (car assert)))
         (write "Errore nella formula atomica" assert _numb)
                                          (*throw 'first_level))
        (t (let ((predicate_name (predicate_of assert)))
                (cond ((not (member predicate_name predicates))
                       (setq predicates (cons predicate_name
                                          predicates))))
                (putprop predicate_name
                  (append (get predicate_name 'axiom) (list assert))
                    'axiom)))))

(def predicate_of (macro (assert) '(cond ((atom (car ,(cadr assert)))
                                          (cadr ,(cadr assert)))
                                    (t (caar ,(cadr assert)))]



(defun clear_all ( )
  (clear_funct functions)
  (clear_vars variables)
  (clear_pred predicates))

(defun clear_funct (funct_list)
  (map_null funct_list 'rm_funct))
(defun rm_funct (funct_name)
  (setq functions (delete funct_name functions))
  (putprop funct_name nil 'funct))

(defun clear_vars (vars_list)
  (map_null vars_list 'rm_vars))
(defun rm_vars (vars_name)
  (setq variables (delete vars_name variables))
  (putprop vars_name nil 'variable))

(defun clear_pred (pred_list)
  (map_null pred_list 'rm_pred))
(defun rm_pred (pred_name)
  (setq predicates (delete pred_name predicates))
  (putprop pred_name nil 'axiom))

(def show (nlambda (pred_names)
  (until (null pred_names)
         (map_null (get (nextl pred_names) 'axiom) 'print))))

(def vars (nlambda (list)
  (until (null list)
         (cond ((not (member (car list) variables))
                (setq variables (cons (car list) variables))))
         (putprop (nextl list) t 'variable))))

(defun map_null (list funct)
  (cond ((null list) nil)
         t (funct (car list)) (map_null (cdr list) funct))))

; la funzione "attach" permette di definire quale sono le i predicati
; a cui sono associate delle funzioni

(def attach (nlambda (func_list)
     (cond ((not (member (car func_list) functions))
            (setq functions (cons (car func list) functions))))
            (setq functions (cons (car func list) functions))))
     (putprop (car func_list) (cadr func_list) 'funct)))

; la "attach" stessa è poi usata per costruire alcuni predicati/funzione
; predefiniti come la "set", "not", "assert", "m_assert", "delete",
; "m_delete"
```

```
(attach set (lambda (x1 y1)
     ;posso fare degli assegnamenti solo a delle variabili unbound
          (let ((lisp_value (eval y1)))
               cond ((or (not (atom x1))
                         (numbp x1))
                     (write "error in set!!!")
                     nil)
                    ((get x1 'variable)
                     (setq save (cons (cdar stack) save))
                     (rplaca (cdar stack) (cons (cons x1 (cons
                                                  lisp_value '(( ))))
                           (cadar stack)))
                     t)
                    (t nil)))))

(putprop '/ '((/)) 'axiom)
(putprop 'rread '(((rread x) ‹ (set x (read)))) 'axiom)

(attach not (lambda (x)
                 (let ((int_mode 'first))
                      (not (?? x))))))

(attach m_assert (nlambda (x)
                  (map_null x (get 'assert 'funct) ) t))

(attach delete (lambda (x) (deli x)))

(defun deli (x)
  (prog (j)
    (setq j nil)
    (return (cond ((setq j (get (caar x) 'axiom))
                   (putprop (caar x) (del (car x)j) 'axiom)

(defun del (x j)
  (cond ((null j) nil)
        ((veq x (caar j)) (del x (cdr j)))
        (t (cons (car j) (del x (cdr j))))))
(defun veq (q f)
  (cond ((equal q f) t)
  (cond ((atom q) (variab q)); !*qui dovrei dare a q il valore di t
                                                    nella blist
  (cond ((veq (car q) (car f)) (veq (cdr q) (cdr f)))))
(def variab (macro (x)
  `(and (atom. (cadr x)) (not (numbp .(cadr x))) (get .(cadr x)
                                                    'variable)))]

(attach assert (lambda) (x)
                 (car (putprop (caar x)
                               (cons x (get (caar x) 'axiom))
                               'axiom)) t))

(attach m_delete (nlambda (x) (map_null x 'deli) t))

(def nextl (macro (x)
  `(let ((y (car .(cadr x)))) (setq .(cadr x) (cdr. (cadr x))) y)))
```

UN LABORATORIO PER LA COMPRESSIONE DI TESTI CON TECNICA WORD MAPPING

O. Bosatelli, M. Bosetti
Istituto di Cibernetica - Università degli Studi di Milano

RIASSUNTO

In questo lavoro viene presentato un laboratorio per la compressione di testi in ambiente Unix. In una prima parte viene fornita una panoramica delle tecniche di compressione correnti e in una seconda viene descritta la nostra implementazione, che ricorre a tecniche di word-mapping. Infine, sono esposti i dispositivi end-user che compongono il laboratorio.

ABSTRACT

In this paper, a laboratory for text compression in Unix environment is presented. First we survey the current compression techniques and then we describe our implementation based on word-mapping. Finally the end user tools that compose the laboratory are outlined.

1. INTRODUZIONE

Malgrado la disponibilità sempre maggiore di memorie di alta capacità e a basso costo, lo studio di tecniche per la compattazione di testi continua ad avere un notevole rilievo. Se, infatti, si inizia a pensare ad un libro elettronico come ad una realtà, parallelamente vanno pensate adeguate tecniche di compattazione per minimizzare l'ingombro in termini di memoria di massa ed ottimizzare tempi e costi di trasmissione tuttora alti sulle odierne reti di comunicazione. Questi benefici sono evidentemente bilanciati da una attività di elaborazione necessaria ad effettuare la compressione e decompressione del testo in oggetto e, pertanto, parametri qualificanti per le diverse tecniche risultano essere il rapporto di compressione ed il tempo di calcolo richiesto.

In generale alla base di ogni algoritmo di compressione vi è il tentativo di individuare e sostituire elementi del testo (caratteri, stringhe, parole) con rappresentazioni codificate in forma compatta mantenendo inalterata la quantità di informazione contenuta nel testo stesso. Nel seguito di questo lavoro vengono passate in rassegna le tecniche disponibili in questo ambito e viene descritta una implementazione basata su word mapping con i relativi tools messi a disposizione dell'utente finale.

2. TECNICHE DI COMPRESSIONE

Nella presentazione dello stato dell'arte sulle tecniche di compressione si è pensato di classificarle in tre gruppi. Per ogni gruppo sono qui riportate le caratteristiche salienti senza entrare in dettagli implementativi che sono disponibili nella bibliografia referenziata.

Il primo gruppo è denominato *statical encoding* [11] e si basa sui codici di Huffman [14],[19].
Viene riconosciuta come entità base il carattere a cui viene assegnato un codice di codifica diverso dall'ASCII con un numero fisso di bit. Per ottenere una rappresentazione codificata per ogni carattere, si considera la frequenza con cui lo si incontra in un certo testo e si assegna ad esso un codice la cui lunghezza è inversamente proporzionale all'occorrenza.

In tale operazione è ovviamente richiesta l'univoca decifrabilità, cioè la capacità di riottenere il carattere originale dopo una operazione di decodifica. I codici in questione, per come sono stati costruiti, sono a lunghezza variabile ed è necessario che soddisfino alle seguenti condizioni:

proprietà del prefisso

dove nessun codice β_i è un prefisso di un altro codice β_j, ovvero nessun codice deve essere contenuto nella parte iniziale di un altro.

completa reversibilità

capacità di ricostruire il testo originale da quello codificato con un numero finito di passi. Questo tipo di tecnica ha il vantaggio di essere del tutto generale ed indipendente dalla semantica del testo.
Un ulteriore guadagno può essere ottenuto in connessione con l'eliminazione dei blank non significativi nel testo.

Un secondo gruppo è denominato di *pattern encoding*. Un pattern è definito in generale come una sequenza generica di caratteri alfanumerici.
Tutte le tecniche di pattern encoding si differenziano solo nelle caratteristiche, che per dimensione e tipo, dei patterns considerati. In questo ambito vi è un approccio che tende a compattare solo una parte del testo in esame [21] ed un altro che tende a codificare completamente il testo sorgente [17],[6].
Quando si procede alla codifica con una tabella di pattern, le cui combinazioni non coprono l'intero universo linguistico, lo si fa perchè quel numero ristretto di pattern ricorre frequentemente o è legato alle particolarità dei campioni di testo. Si pensi a tali tecniche applicate ai *business file* [22], alla *differencing* [11] [18] ed anche ai *digrafi* [7] stilati secondo frequenza.

L'applicabilità di una tecnica che studia e raccoglie un insieme di pettern, che siano chiusi rispetto alla lingua in esame, è molto ampia anche se i risultati, in termini di tasso di compressione, non sono nel complesso rilevanti. Queste tecniche vengono considerate positivamente quando si vuole compattare file particolari, con alcune regolarità che si possono riconoscere a priori o si possono desumere da studi statistici.

Una estensione del pattern encoding è quella di scegliere come minima entità logica la *parola,* definibile come una sequenza di caratteri delimitati da un insieme di end-chars (spazi bianchi, segni di interpunzione, ecc.)

Il terzo ed ultimo gruppo è appunto basato su tecniche *word-mapping.*
Queste tecniche sono applicabili, in generale, a testi di una stessa lingua con un buon tasso di compressione. L'algoritmo che viene qui presentato rientra appunto in tale categoria.

3. WORD MAPPING

L'algoritmo implementato è orientato alla compattazione di testi leggerari e/o tecnici in lingua italiana con un tasso di compressione che varia da 35 a 50 per cento in qualsiasi testo di tali categorie.
Pertanto da uno studio volto a individuare le regolarità e le eccezioni della lingua italiana [2] in un corpus campione, è possibile costruire un dizionario contenente i termini più ricorrenti che quindi saranno codificati. Un dizionario così costruito è di carattere generale, cioè contiene parole di normale utilizzo in ogni tipo di testo. È possibile, però, anche costruire dizionari specifici il cui contenuto è stabilito di volta in volta a partire dal singolo testo.

L'algoritmo in oggetto prevede l'utilizzo di un dizionario di 1024 voci globali diviso in due parti: una di carattere generale contenente 850 parole italiane, le più frequenti secondo le liste del LIF (Liste Italiane Frequenza) [2], e di una seconda di 173 termini attivata dinamicamente in base ad una selezione dell'utente. Quest'ultima, al momento, può essere o di contenuto scientifico-informatico o costuita da termini estratti da testi di letteratura e quotidiani.

Le 1024 voci che costruiscono il dizionario sono rappresentate solo dalla loro *radice* poichè viene prima effettuata una elaborazione al fine di eliminare le desinenze di flessione [8].

La struttura dati utilizzata per il dizionario delle radici è una lista doppiamente segmentata [25]; i termini sono ordinati secondo due criteri: per lunghezza della radice in caratteri e nell'ambito di questo gruppo per ordine alfabetico.

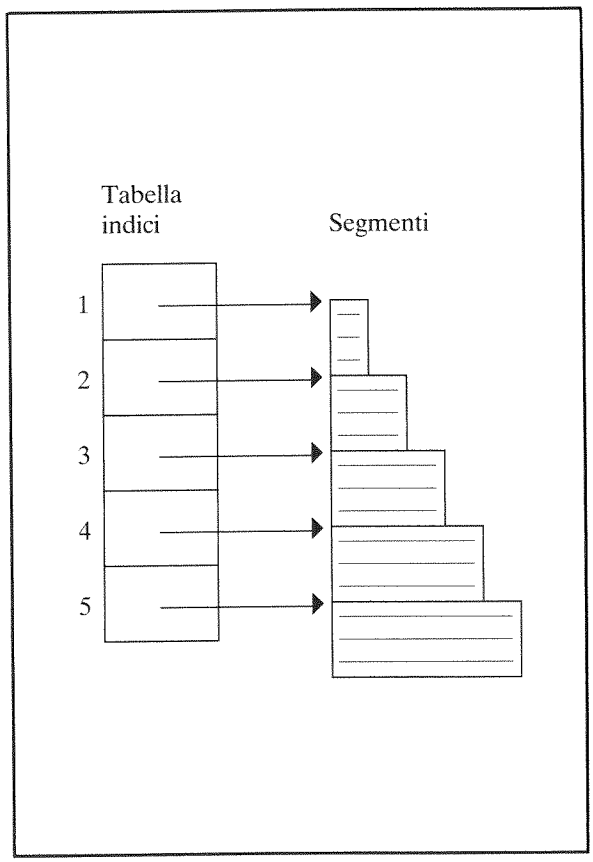

Fig. 1

In fase di ricerca dopo un test sulla lunghezza della radice si trova il giusto segmento e su questo si effettua la ricerca dicotomica. Nel caso in oggetto il più lungo segmento è costruito da circa duecento parole, quindi gli accessi alla struttura dati per trovare un termine fra gli 850 del dizionario generale sono contenuti anche nel peggiore dei casi.

4. COMPRESSIONE DEL TESTO

L'idea di base consiste nel rimpiazzare nel testo le parole più ricorrenti, cioè reperibili in uno dei due dizionari (generale, specifico, con un codice che rappresenta la posizione di quel termine nel dizionario stesso). Il codice è di due bytes, per ogni parola codificabile, ed è così costruito:

I dieci bits meno sifnificativi sono per il codice della radice cioè la posizione nel dizionario dove tale termine è memorizzato in forma estesa; altri cinque bits identificano il codice della desinenza (sono stati trattati 32 tipi di desinenze). Il bit iniziale serve come indentificatore e consente di decidere se i seguenti due bytes contengono un codice o meno. Pertanto, in fase di decodifica, il flag non impostato rivela la presenza di caratteri Ascii che, come tale, vien trascritto in uscita inalterato, mentre, se si è in presenza di un codice, viene attivata una procedura che interpreta i patterns di bits secondo la struttura prefissata risalendo alla parola originaria.

L'utente, all'atto della chiamata del compattatore, specifica come parametro, quale dizionario secondario intende usare e se desidera al termine dell'esecuzione della routine una valutazione in percentuale del tasso di compressione e/o una valutazione in forma grafica del tempo impiegato per ottenere il testo compattato.

Sulla base di una analisi con testi campione, si è verificata una dipendenza quasi lineare del tempo di CPU dalla dimensione espressa in caratteri del testo.

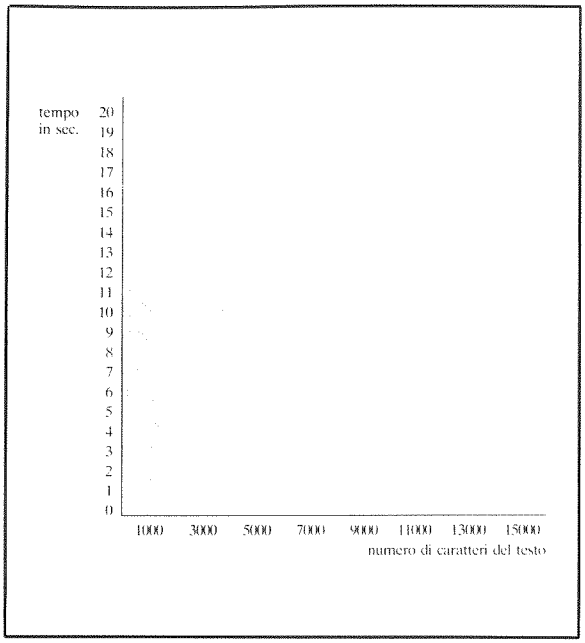

Fig. 3

5. IL LABORATORIO

Nell'affrontare il problema della compattazione dei testi nell'ambito del Sistema Operativo Unix si è ritenuto opportuno implementare un ambiente globale per la analisi di testi che si è denominato *laboratorio*.

Scopo principale dei moduli attualmente implementati è quello di fornire una serie di informazioni sul testo in oggetto orientate ai problemi di compressione. Il laboratorio costituisce un package evidentemente ampliabile nel tempo con una serie di tools rivolti all'utente finale che consentono quindi di affrontare in modo sistematico e da diversi punti di vista lo studio dei testi.

Tornando al problema della compattazione, le infor-

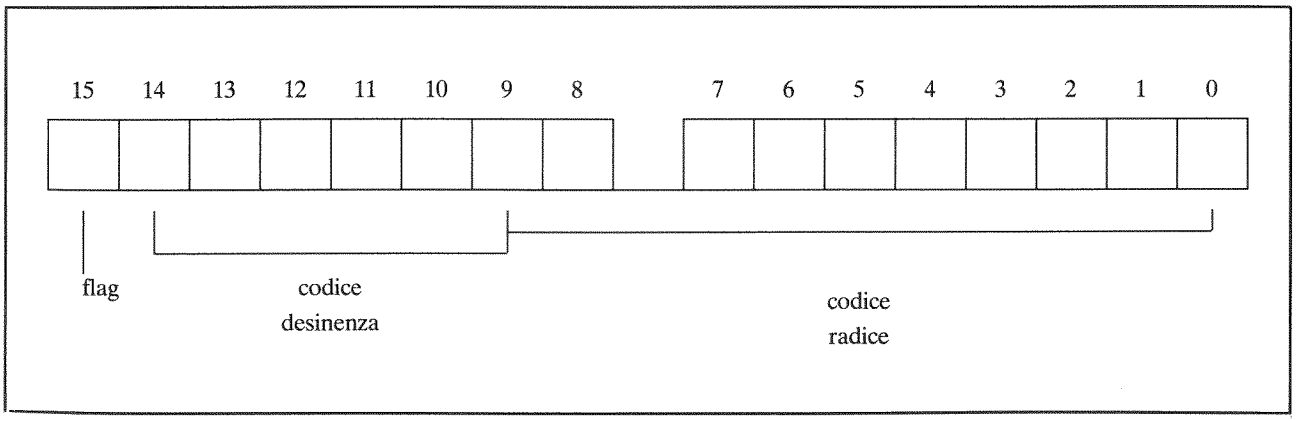

Fig. 2

mazioni prodotte sono di carattere tutto generale e prescindono dalla particolare tecnica di compressione implementata e descritta in precedenza.

L'approccio utilizzato per individuare e poter quantizzare gli elementi caratteristici del testo è quello della statistica linguistica. Un testo, infatti, può esse-

re considerato come un insieme di vocaboli distribuiti secondo leggfi statiche. In tale ambito, un primo obiettivo consiste nello studiare la stabilità della frequenza con cui vengono adoperati i diversi termini nel linguaggio scritto. Si è dimostrato che, per quanto riguarda la ripartizione delle parole, un numero relativamente piccolo di vocaboli molto frequenti, costituisce la maggior parte dell'intero testo: i 15 vocaboli più frequenti rappresentano circa un quarto del testo, i primi 100 il 60.
Diviene allora possibile organizzare le parole disponendole in ordine decrescente secondo la loro frequenza di apparizione e creando così per le diverse lingue naturali dei *dizionari di frequenza.*

Il numero d'ordine del posto occupato da una graduatoria così formata si definisce rango di quella parola. Secondo la *legge di Estoup* il prodotto fra rango e frequenza di una parola è costante. Questa legge è tanto più esatta quanto più è grande il numero N di parole considerato. Il valore costante di questo prodotto è indicabile con C (N). In base a tale legge si può scrivere:

$$1^{*}f_i = c(N) = 2^{*}f_2 = c(N) = \ldots = N^{*}f_N = c(N)$$

con f_i frequenza assoluta della parola di rango i e con N rango massimo.

Ecco un esempio dei valori ottenuti su un testo di 60000 parole [23]:

rango	frequenza (assoluta)	(rango) x (frequenza)
10	2653	26530
100	265	26500
1000	26	26000
12000	2	24000
29000	1	29000

In un sistema di coordinate cartesiane, con il rango in ascissa e la frequenza in ordinata, la curva che si ottiene è un'iperbole equilatera.

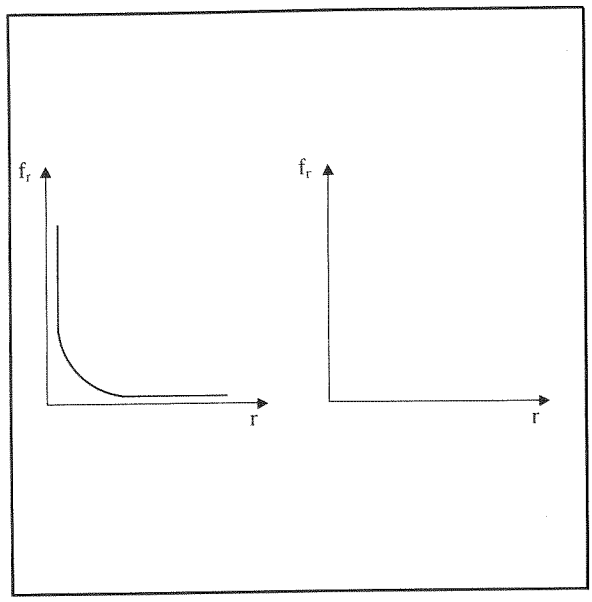

Fig.4

Il grafico di destra è equivalente, ma in scala logaritmica. Tali rilevazioni trovano poi formalizzazione nelle *leggi di Zipf e Mandelbrot* che sono alla base di numerosi studi nell'ambito della statistica linguistica.

Lo scopo dell'implementazione del laboratorio consiste nel fornire all'utente finale informazioni di carattere statistico sul testo che intende analizzare.
Vengono fornite quindi le liste di frequenza dei singoli caratteri, delle parole del testo, ed il diagramma di Estoup.

Si può conoscere inoltre il numero di occorrenza e quindi la frequenza relativa di ogni elemento di testo, la lunghezza media in caratteri di ogni parola, il numero di parole in base alla lunghezza. Questi risultati sono ottenibili sul singolo testo; se si vogliono generalizzare i risultati occorre porsi il problema della rappresentatività del campione per quanto riguarda il numero e le caratteristiche dei testi da prendere in esame per ottenere una lista attendibile.

Altri risultati forniti sono finalizzati all'algoritmo di compressione basato sulla tecnica word-mapping, e riguardano, quindi l'unità logica parola.
Si calcola il numero e la percentuale delle parole di testo che sono anche presenti nei vari dizionari utilizzati ed è presentata la stima del tasso di compressione in funzione del dizionario scelto. Ciò fornisce uno strumento per valutare la convenienza ad utilizzare l'algoritmo di compressione e con quali parametri chiamare la primitiva di compressione. Accanto a questi risultati, sono fornite altre valutazioni in forma grafica.

Un primo diagramma mette in relazione il tasso di compressione ottenuto con la percentuale dei caratte-

ri codificati. Si verifica sperimentalmente che anche con una compressione parziale del testo si ottiene un tasso paragonabile ad altre tecniche che coinvolgono la codifica completa del testo.

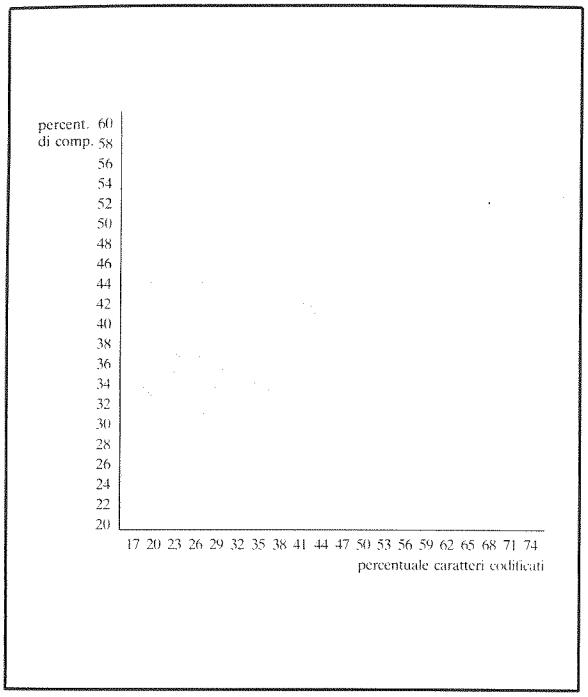

Fig. 5

La figura 6 mostra invece, un grafico riepilogativo perchè fornisce una situazione a fronte di un certo insieme di testi sottoposti a compressione.

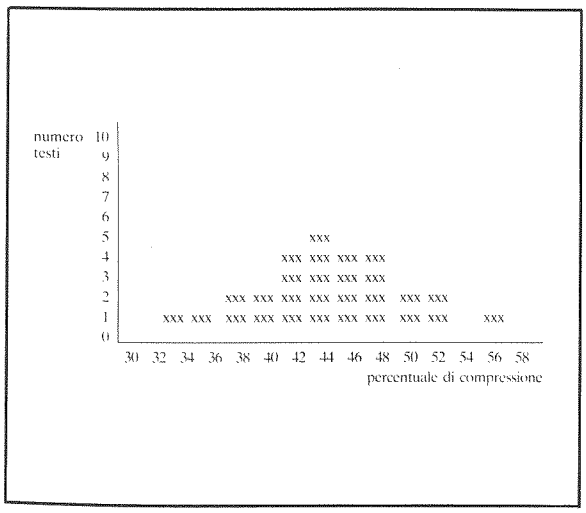

Fig. 6

6. CONCLUSIONE

Nel presente lavoro è descritto un approccio al problema della compressione tecnica word-mapping basato sulla consultazione di dizionari di frequenza. Sono stati inoltre presentati una serie di tools per l'utente finale che pongono le basi per la costruzione, in prospettiva, di un laboratorio esteso per applicazioni nell'ambito del text processing.

Questo lavoro si è svolto nell'ambito di una attività di tesi [3] [4] per il Corso di Laurea in Scienze dell'Informazione di Milano. L'implementazione è stata realizzata ed è disponibile sull'elaboratore Onix C-8000, con Sistema Operativo Unix installato presso l'Istituto di Cibernetica.

7. RINGRAZIAMENTI

Si ringrazia il dr. Dario Lucarella per il supporto fornito durante l'intera attività e per i suggerimenti alla stesura del presente articolo.
Un grazie particolare al professor Gianni Degli Antoni per il contributo originale al contenuto della nostra tesi.

8. BIBLIOGRAFIA

[1] S. Akl, ON THE SECURITY OF COMPRESSED ENCORDINGS, Queen's University Kingston, Canada Technical report nr. 83-151, august 1983

[2] U. Bartolini, C. Tagliavini, A. Zampolli, LESSICO DI FREQUENZA DELLA LINGUA ITALIANA CONTEMPORANEA, IBM Italia, 1971

[3] O. Bosatelli, COMPRESSIONE DEI TESTI, Università degli Studi di Milano, Corso di Laurea in Scienze dell'Informazione, dicembre 1984

[4] M. Bosetti, LABORATORIO PER LO STUDIO DI PROBLEMI DI COMPATTAZIONE, Università degli Studi di Milano, Corso di Laurea in Scienze dell'Informazione, dicembre 1984

[5] R.M. Capocelli, ON SOME FREE SUBSEMIGROUPS OF A FREE SEMIGROUP, Università di Salerno, Semigroup Forum 15 pp. 125-135, 1977

[6] R.M. Capocelli, L.M. Ricciardi, A HEURISTIC APPROACH TO FEATURE EXTRACTION AND COMPRESSION FOR WRITTEN LANGUAGES, IAC "Mauro Picone" pubblicazione serie 3 nr. 125, Roma 1978

[7] D. Cortesi, AN EFFECTIVE TEXT-COMPRESSION ALGORITHM, Byte, p.p. 397-403, jan 1982

[8] M. Del Canto, UN MODELLO DI RAPPRESENTAZIONE DEL LESSICO PER LA ELABORAZIONE AUTOMATICA DELLA LINGUA ITALIANA, Elettronica San Giorgio, Elsag, 1982

[9] A. Fraenkel, M. Mor, Y. Pearl, IS TEXT COMPRESSION BY PREFIXES AND SUFFIXES PRACTICAL, Weizmann Institute, Israel Acta Inf. vol. 20 nr. 4 pp. 371-389, 1983

[10] R. Gallagher, VARIATIONS ON A THEME BY HUFFMANN, IEEE Trans. vol. it-24 nr. 6 pp. 668-674, nov 1978

[11] D. Gottlieb, S. Hagerth, P. Lehot, H. Rabinowitz, A CLASSIFICATIONS OF COMPRESSION METHODS AND THEIR USEFULNESS FOR A LARGE DATA PROCESSING CENTER, Fireman Fund American Insurance Companies, National Comp. Conference (44) pp. 453-458, 1975

[12] B. Hahn, A NEW TECHNIQUE FOR COMPRESSION AND STORAGE OF DATA, University of Waterloo, Comm. of the ACM vol. 17 nr. 8, aug 1974

[13] H. Hou, DIGITAL DOCUMENT PROCESSING, Ph. D. Xerox Corporation, John Wiley and Sons cap. 5 pp. 116-121, nov 1983

[14] D. Huffmann, A METHOD FOR THE CONSTRUCTION OF MINIMUM-REDUNDANCY CODES, Associate IRE, Proc. IRE vol. 40 nr. 9 pp. 1098-1101, set 1952

[15] R. Lea, TEXT COMPRESSION WITH AN ASSOCIATIVE PARALLEL PROCESSOR, Brunel University Middlesex, The Computer journal vol. 21 nr. 1 pp. 45-56, feb 1978

[16] A. Lempel, J. Ziv, ON THE COMPLEXITY OF FINITE SEQUENCES, Member IEEE, IEEE Trans. Infor. theory vol. it-22 pp. 75-81, jan 1976

[17] A. Lempel, J. Ziv, A UNIVERSAL ALGORITHM FOR SEQUENTIAL DATA COMPRESSION, Member IEEE, IEEE Trans. Infor. theory vol. it-23 pp. 337-343, may 1977

[18] A. Marini, P. Slocovich, INDICIZZAZIONE DI DOCUMENTI, Università di Milano, Sistemi infor. industriali

[19] M. Pechura, FILE ARCHIVAL TECHNIQUES USING DATA COMPRESSION, Comm. of the ACM vol. 25 nr. 9 pp. 605-609, sept 1982

[20] M. Rodeh, V. Pratt, S. Even, LINEAR ALGORITHM FOR DATA COMPRESSION VIA STRING MATCHING, Ibm Israel-Mit-Haifa center, Journal of the association for computing machinery vol. 28 nr. 1 pp. 16-24, jan 1981

[21] F. Rubin, EXPERIMENTS IN TEXT FILE COMPRESSION, IBM Poughkeespie USA, Comm. of the ACM vol. 19 nr. 11 pp. 617-622, nov 1976

[22] S. Ruth, P. Kreutzer, DATA COMPRESSION FOR LARGE BUSINESS FILES, Datamation pp: 62-66, sept 1972

[23] F. Serbanescu, CONVERSAZIONI SULLA STATISTICA LINGUISTICA, IBM Italia, Ed. tecnico scientifica Pisa, 1969

[24] R. Tropper, BINARY-CODED TEXT: A TEXT-COMPRESSION METHOD, Rhode Island College, Byte pp. 398-413, apr 1982

[25] T. Turba, CHECKING FOR SPELLING AND TYPOGRAPHICAL ERRORS IN COMPUTER-BASED TEXT, Sperry Roseville Development center, Comm. of the ACM pp. 51-60, 1981

[26] R. Wagner, COMMON PHRASES AND MINIMUM-SPACE TEXT STORAGE, Cornell University, Comm. of the ACM vol. 16 nr. 3, mar 1973

[27] H. White, PRINTED ENGLISH COMPRESSION BY DICTIONARY ENCODING, Member IEEE, Proc. of the IEEE vol. 55 nr. 3 pp. 390-396, mar 1967

UN SISTEMA PER LA DOCUMENTAZIONE DI PROGRAMMI DIDATTICI

Marta Ferrari - Le Van Huu
Istituto di Cibernetica, SiLAB - Università degli Studi di Milano

RIASSUNTO

MIDA è un sistema di documentazione che fornisce meccanismi utili sia per il reperimento sia per la comprensione. La decisione di costruire un sistema di documentazinoe con capacità di browsing è nata dall'osservazione delle difficoltà incontrate presso il nostro Laboratorio (siLAB, Università di Milano) nell'organizzare e classificare un rilevante numero di programmi didattici. Ciò ha comportato una scarsa visibilità da parte degli utenti, principalmente studenti e docenti, delle informazioni relative ai lavori svolti da altri.
Il sistema comprende una metodologia che permette agli studenti di stendere la documentazione dei loro programmi, specificandone le caratteristiche secondo uno schema prefissato.
I base a tale documentazione, ogni programma può essere catalogato automaticamente e quindi reperito dall'utente mediante un semplice ed immediato meccanismo di ricerca, pur partendo da ridotte informazioni iniziali (browsing).

ABSTRACT

MIDA is a documentation system that provides mechanism for both retrieval and comprehension needs.
The decision to design and implement a documentation system with browsing capabilities was made when we noticed in our Laboratory (siLAB, University of Milan) the difficulties encountered in organizing and classifying a consistent number of teaching programs. This has led to poor visibility of work and relative information on the part of users, mainly students and teachers, other than their own.
This system includes a methodology which allows students to make the documentation of their programs, specifying their characteristics according to pre-defined groups.
On the basis of this documentation, every program can be catalogued automatically and then retrieved by the user by means of a simple and immediate search mechanism; even if the initial information was limited (browsing).

1. INTRODUZIONE

Nell'ambito del Corso di Laurea in Scienze dell'Informazione dell'Università degli Studi di Milano le esperienze pratiche di programmazione si svolgono presso il Laboratorio Didattico. Gli studenti dispongono sin dal primo anno di tempo macchina per svolgere le esercitazioni didattiche relative ai vari corsi oltre che per preparare e sostenere esami e tesi di laurea.
In particolare, nel primo biennio l'uso del Laboratorio tende ad insegnare ed esaltare la capacità di progettare, costruire e presentare programmi di dimensione significativa insieme a quella di definire le specifiche di piccoli sistemi, da sviluppare con pesante reimpiego di software già disponibile.

In ambedue i casi, obiettivo irrinunciabile è stata la dimensione del progetto, sufficientemente ampia da non rendere "finzione" le attività di specificazione e di sviluppo, ma anzi da renderle assolutamente realistiche: una condizione inevitabile a tale proposito è stata quella dell'organizzazione del lavoro in gruppi di progetto; ciascuna delle terne in cui gli studenti sono stati suddivisi è stata un'unità di sviluppo nel cui ambito si sono discusse e definite le specifiche funzionali, si è suddiviso il progetto in componenti, si è distribuito il lavoro in gruppi di sviluppo separati.
La decisione di rendere disponibile l'utilizzo degli elaboratori fin dal primo anno si basa sull'esigenza di procedere alla alfabetizzazione in informatica degli studenti, al fine di accelerare il processo di apprendimento dei concetti di base.

L'esperienza compiuta ha confermato la validità dell'approccio ed ha convinto della necessità di poten-

ziare la struttura ed estendere le risorse di calcolo disponibili.
L'elevato numero di studenti (oltre 3000 al primo biennio, oltre 1000 al secondo) e la modalità di uso degli elaboratori (a cui accedono a terne) comportano una produzione consistente di programmi didattici con l'ovvia conseguenza di una pressante necessità di una loro adeguata organizzazione e classificazione.
Con il procedere della vita del Laboratorio si è evidenziato, oltre alle problematiche suddette, na scarsa visibilità da parte degli utenti (studenti e docenti) delle informazioni relative ai lavori svolti da altri con notevoli svantaggi. Infatti gli studenti non sono adeguatamente orientati, soprattutto nelle loro prime esperienze di programmazione, da esempi di programmi già realizzati dai loro colleghi, effettuano inutili ripetizioni di parte di programmi che già ben svolgono le stesse funzioni, sono sprovvisti di strumenti per confrontare le proprie idee architetturali con quelle altrui quindi proporre versioni ottimizzate di algoritmi preesistenti.

D'altra parte, ai docenti mancano elementi per una valutazione più omogenea dei programmi realizzati e per la formulazione di nuovi temi d'esame, anche in relazione a quelli già svolti, per far sì che singoli temi costituiscano componenti di un più ampio progetto.
In sintesi, ciò comporta una inutile duplicazione di sforzi di programmazione da parte degli studenti e una mancata verifica complessiva del lavoro svolto.

Per superare le difficoltà esaminate si è definito e messo a punto un sistema di disseminazione delle informazioni che, attraverso la documentazione di programmi didattici, fornisca un meccanismo sia di reperimento sia di comprensione degli stessi.
Tale sistema comprende una metodologia che permette agli studenti di stendere la documentazione dei loro programmi specificandone le caratteristiche secondo uno schema prefissato con l'uso di termini appartenenti ad un vocabolario specifico estendibile e classificandoli all'interno di gruppi predefiniti.
In base a tale documentazione, ogni programma può essere catalogato automaticamente e quindi reperito dall'utente mediante un semplice ed immediato meccanismo di ricerca, pur partendo da ridotte informazioni iniziali (browsing).

Inoltre il sistema permette l'analisi statistica dell'attività degli studenti nell'ambito del Laboratorio (numero di adesioni a temi d'esame, tempi di accesso...), la valutazione e l'utilizzo del prodotto finale (definizione della struttura del programma a partire dalle asserzioni, confronto fra più programmi dello stesso tipo, utilizzo comune di programmi ben documentati e di buona qualità, produzioni di raccolte annuali di esercizi svolti e temi d'esame...), l'amministrazione faciltata di archivi (di programmi) di dimensioni considerevoli e in rapida evoluzione.

2. DESCRIZIONE DEL SISTEMA

Il sistema denominato MIDA (Metodologia Interattiva di Documentazione Automatica), è stato sviluppato mediante il linguaggio di programmazione Pascal-UCSD in ambiente UNIX°, i cui files sono organizzati in una struttura ad albero. In tale struttura ogni nodo non terminale rappresenta una direttrice contenente i nomi dei nodi sottostanti. I nodi terminali rappresentano, invece, i files.

2.1 Organizzazione del sistema

L'elemento base reperibile all'interno del sistema è il file ordinario che può essere una procedura, un programma sorgente, un oggetto eseguibile, un testo, ecc.. Definiamo tale elemento "documento". Ogni documento per essere classificabile deve contenere, oltre al testo effettivo, informazioni aggiuntive utili per la sua catalogazione automatica. Tali informazioni sono distinguibili dalla parte testo del documento mediante marcature apposite:

.TL : titolo
.AB : abstract
.FL : documenti collegati al Documento in questione
.MA : manuale
.IN : termini di indicizzazione
.AU : autore
.BI : bibliografia
.DA : data
.CO : corso
.DO : docente

I documenti sono identificabili da una o più parole chiave (argomento), cioè quelle parole contenute in particolari sezioni di documentazione del documento precisamente .TL, .AB, .IN, che appartengono ad un vocabolario specifico.
L'insieme delle parole chiave è organizzato in una struttura gerarchica ad albero, in cui ogni nodo rappresenta una parola chiave i cui figli sono costituiti da altre parole chiave e/o documenti. Una stessa parola chiave può trovarsi in nodi differenti permettendo di essere raggiunta attraverso cammini alternativi. La parola assume comunque significato differente a secondo del contesto in cui si trova, contesto che è rappresentato dal cammino per essere raggiunta. I documenti, che costituiscono i nodi terminali della struttura, possono comparire in più nodi.

° UNIX è un Trade Mark dei Bell Laboratories della AT&T

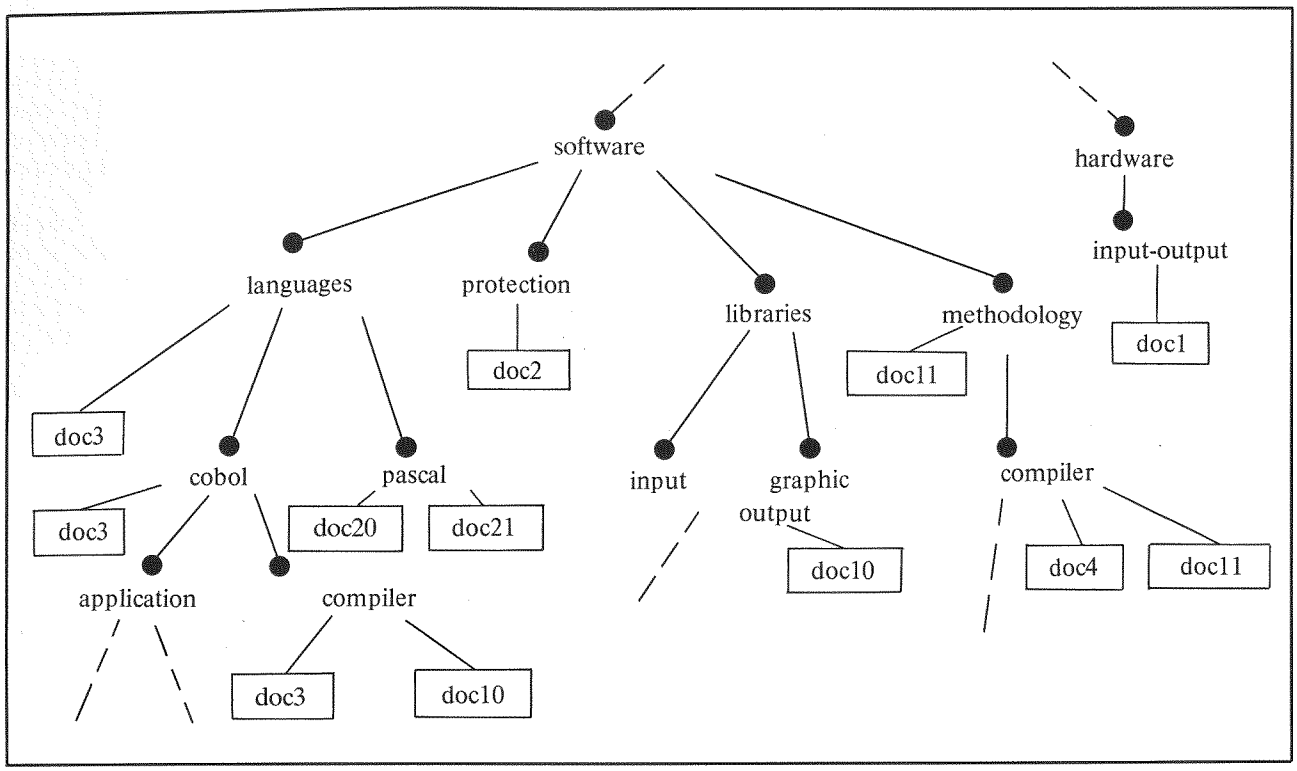

Fig. 1

Una esemplificazione di quanto esposto è riportata in figura 1 in cui i nodi delle parole chiave sono raffigurati dei cerchi, mentre i rettangoli rappresentano i documenti appesi ai nodi stessi. Dall'esempio si può notare innanzitutto che la stessa paola chiave "compiler" trovandosi in più nodi identifica contesti differenti. Il primo contesto, raggiunto mediante il cammino /software/languages/cobol/compiler, siriferisce ad un ambiente relativo ai compilatori specifici per il linguaggio cobol, mentre il secondo contesto (software/methodology) si riferisce a metodologie di sviluppo di software e potrebbe identificare, in modo particolare, documenti relativi alle tecniche di costruzione di compilatori.
L'esempio evidenzia inoltre l'altra caratteristica dell'albero delle parole chiave, cioè quella di permettere allo stesso documento di essere collocato in nodi differenti. Infatti il documento "doc3" (rappresentante probabilmente un compilatore di Cobol) può essere reperito sia mediante la parola chiave "languages" che "cobol" che "compiler".

2.2. Funzionalità

In un ambiente condiviso, quale il Laboratorio Didattico di Scienze della Informazione, per ogni sistema è necessario individuare livelli di accesso differenziati a secondo del tipo di utente.
Anche MIDA (Metodologia Interattiva di Documentazione Automatica) è predisposto di funzionalità differenti per soddisfare le esigenze di diverse categorie di utenti quali:

a. - il gestore del sistema il cui principale compito è la manutenzione dell'albero della parole chiave (operatore)

b. - i gestori dei documenti il cui compito è la preparazione e l'archiviazione degli stessi (proprietari dei documenti, studenti, docenti...)

c. - l'utente che perlustra il sistema partendo da informazioni anche limitate per reperire utili documenti (studenti, docenti, visitatori, curiosi...).

Riportiamo le principali funzionalità delle varie categorie di utenti.

1) *Gestione del Sistema: funzionalità riservate all'operatore*

Queste funzionalità consentono la preparazione e l'ottimizzazione della Base-Dati necessaria per l'utilizzo del Sistema.
È inoltre possibile definire legami di natura semantica tra gli elementi appartenenti alla Base-Dati per mezzo di relazioni di Sinonimia. In particolare all'operatore è fornito un insieme di funzioni per lavorare

nei seguenti ambienti:

a) Costruzione della Struttura-Argomenti
All'interno di questo ambiente sono disponibili tutti i comandi necessari sia per la creazione di un albero n-ario, cioè comandi di cancellazione ed inserimento di nodi con le loro etichette, che per il libero percorrimento di tale struttura.
Alle etichette di ogni nodo può inoltre essere associato un commento che indichi la particolare accezione con cui il termine (parola chiave) etichettante il nodo viene inteso.

b) Gestione delle Sinonimie
Tale ambiente consente la definizione o la rimozione di legami tra un qualsiasi termine ed uno degli Argomenti della Base-Dati.
In generale si avrà perciò la produzione di una tabella di relazioni del tipo:

Sinonimo===) Argomento

dove tale legame è funzionale, cioè più Sinonimi possono essere collegati allo stesso Argomento, ma non viceversa.
In sostanza, dopo una tale definizione, indicare la parola definita come sinonimo equivarrà ad indicare l'argomento.

c) Ottimizzazione delle Strutture di Base
Questa operazione consiste nel bilanciamento degli alberi componenti la struttura interna del sistema, consentendo riduzioni sul costo delle ricerche dell'ordine del 15-30%.

2) *Gestione dei Documenti: Funzionalità Riservate all'Utente*

In un ambiente del tipo "laboratorio informatico", un progetto preparato, verificato completato e considerato come corretto, deve essere corredato della sua documentazione e quindi successivamente archiviato.
Queste funzionalità riguardano da vicino quegli utenti che sono i proprietari dei documenti costituenti la Base-dati del sistema. Ai proprietari dei documenti è riservato un insieme di funzioni atte alla gestione dei loro documenti, come la possibilità della composizione automatica di documenti con l'analisi immediata del contenuto, oppure l'archiviazione, il reperimento, e la comprensione di testi.
Le principali funzioni sono:

a) Composizione Interattiva della Documentazione
Tale funzione viene realizzata per mezzo di un semplice compositore, il quale permette di inserire in Documenti già esistenti o creati al momento, parti di testo denominate "periodi".
I periodi così composti sono poi analizzati immediatamente e da essi vengono estratte le parti di Documentazione (secondo la metodologia presentata) per essere sottoposte ad analisi successivamente.

A questo punto le parole-chiave eventualmente contenute nelle parti di Documentazione considerate vengono direttamente evidenziate su video e ne è poi fornita la lista, comprensiva anche delle particolari accezioni secondo cui tali termini sono stati interpretati (molto importante per permettere un'archiviazione "per contenuto" piuttosto precisa).
La Funzione di Composizione Interattiva consente di collocare all'interno del Documento (generalmente in testa) alcuni periodi che lo descrivano e che, in quanto immediatamente analizzati, permettano un'archiviazione automatica "integrata" con la fase di editing.
Un'altra (e forse ancor più interessante) possibilità di utilizzo consiste invece nel preparare la Documentazione prima della compilazione del Documento, in modo che tale costruzione, secondo la metodologia proposta, funga anche da direttiva per l'effettivo sviluppo del progetto. Questa idea risulta particolarmente valida anche dal punto di vista didattico in quanto contribuisce ad abituare lo studente a documentare il progetto prima di realizzarlo e quindi a far necessariamente precedere le fasi strettamente realizzative da accurate fasi di disegno.
Si può quindi parlare di vera e popria metodologia.

b) L'archiviazione Automatica Mediante Indicizzazione
Esiste in MIDA la possibilità di Archiviazione "secondo contenuto" di Documenti contenenti "parti di Documentazione", così come specificato nella metodologia presentata.
Una volta controllata la disponibilità del Documento, si procede ad una estrazione della Documentazione potenzialmente significativa per caratterizzare il Documento e ad un confronto fra i termini conosciuti dai sistemi e quelli estratti dal Documento, nell'intento di individuare tra questi ultimi della parola-chiave.
Tali termini estratti vengono a questo punto considerati come "proposte di Archiviazione" esaminabili dall'utente, il quale ne può verificare l'accezione secondo cui sono intesi ed eventualmente eliminarne alcune come "non caratterizzanti adeguatamente il Documento".

3) *Ambiente di Reperimento Documenti*

In questo ambiente operano gli utenti che non sono nè il gestore del sistema nè i produttori di documenti. Infatti le funzioni di questo ambiente sono orientate a chi reperisce delle informazioni già residenti nel sistema, sia in modo automatico , immettendo comandi o richieste e vagliando le successive risposte, sia in modo "manuale", attraverso meccanismi di "browsing" i quali permettono all'utente, soprattutto se inesperto, di ottenere informazioni probabilmente non acquisibili in altro modo, in quanto il meccanismo stesso di "browsing" tende a suggerire all'utilizzatore "come" e "cosa" reperire, aiutandolo in modo

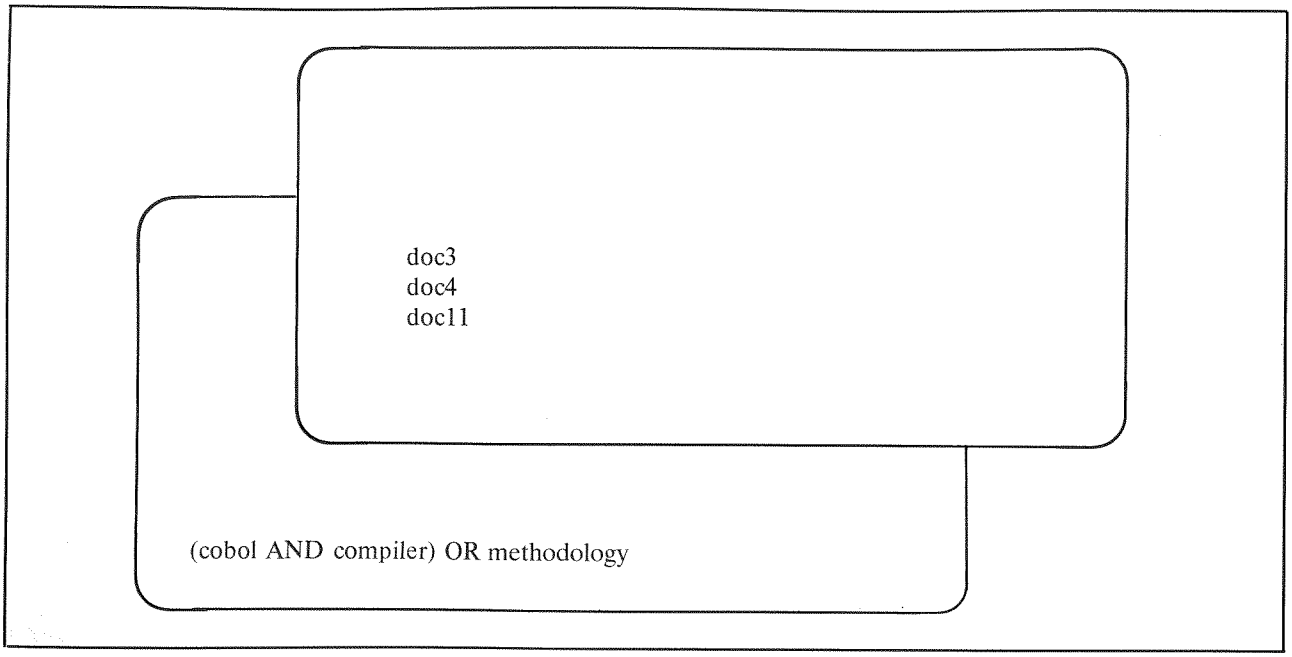

Fig. 2

"naturale" nell'impostazione di ricerche secondo "argomenti". Infatti è possibile effettuare:

a) La ricerca "automatica" di Documenti
Avviene attraverso la specificazione di un'espressione logica composta da parole, da operatori di relazione ("or", "and") e da parentesi le quali permettono espressioni di complessità indefinita.
I documenti il cui contenuto risulti consistente con tale specificazione vengono poi presentati in quanto potenzialmente interessanti e possono essere successivamente prodotti e può essere prodotta la loro sola Documentazione oppure possono essere visualizzate tutte le parole-chiave reperibili al loro interno. Si noti che le parti di Documentazione contenute nel testo vengono fatte procedere da un'opportuna intestazione, nell'intento di facilitarne l'individuazione e la comprensione.
Nel caso la specificazione indicata risulti troppo generica e porti alla produzione di una lista eccessivamente corposa di Documenti, vengono inoltre messi a disposizione meccanismi atti a facilitare la costruzione di ricerche più mirate o secondo Argomenti più specifici.
Il risultato di una ricerca automatica effettuata sull'albero dell'esempio precedente (figura 1) è riportato in figura 2, dove la specificazione "(input-output AND compiler) OR metodology" comporta la selezione dei documenti "doc3", "doc4", e "doc11".

b) "Browsing" dell'Albero-Argomenti
Il meccanismo di "browsing" permette la scansione di una struttura (tipicamente gerarchica) di termini indicanti "materie", "sezioni della conoscenza", "attività",... nell'intento di reperire Documenti che soddisfino certe esigenze informative.
Esistono quindi, in questo ambiente, comandi per il percorrimento dell'albero e l'individuazione della posizione corrente e di possibilità di ulteriore scansione. Inoltre tale meccanismo può essere utilizzato per effettuare Archiviazioni estremamente mirate, individuando un insieme piuttosto preciso di punti di archiviazione. Si ricordi che i documenti reperiti possono essere esaminati secondo i medesimi meccanismi già presentati, i quali sono comuni anche a questo ambiente.
Le figure 3a, 3b, 3c, riportano una sessione d'uso del

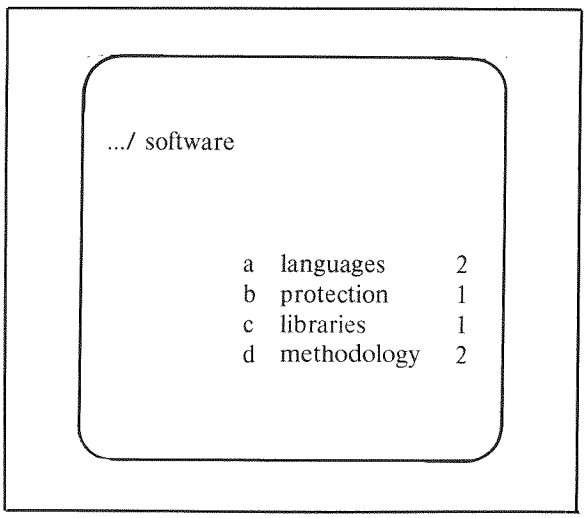

Fig. 3a

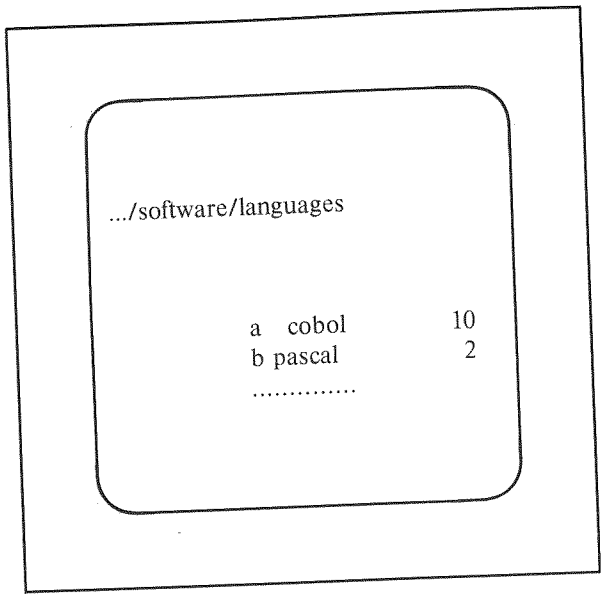

Fig. 3b

meccanismo di browsing. L'utente, a partire da un nodo, per esempio "software", ha a disposizione la lista delle parole chiave del livello immediatamente inferiore (fig. 3a). Accanto ad ogni elemento della lista compare il numero totale di documenti relativi. Selezionando la parola chiave di interesse, per esempio "languages", tutto l'ambiente si sposta al livello inferiore e quindi compare sul video la lista delle parole chiave relative (fig. 3b). Agevolmente l'utente può tornare ai nodi precedenti, oppure ottenere la lista dei documenti relativi al nodo corrente. (fig. 3c).
L'acquisizione di nuove conoscenze lungo questo cammino indirizza l'utente in modo sempre più mirato verso l'informazione di interesse anche se questa non è stata specificata a priori in modo esplicito.

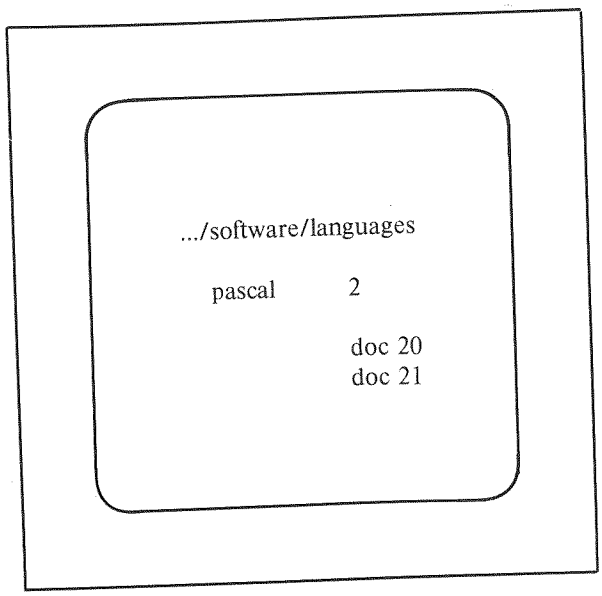

Fig. 3c

c) Visualizzazioni General
Vengono offerte liste consuntive su tutti i possibili contenuti dell'archivio dei Documenti e della base-dati creata dall'operataore.
Tali visualizzazioni, oltre che documentare il tipo di utilizzo del sistema, sia personale che globale, permettono di conoscere la strutturazione dei termini significativi e quindi di effettuare archiviazioni e reperimenti maggiormente consapevoli.

3. CONCLUSIONE

Il sistema presentato oltre ad essere un ottimo strumento per la archiviazione e il reperimento, costituisce senz'altro una valida metodologia di documentazione e di costruzione di programmi.

Usando MIDA il documentare un programma diventa un'operazione sistematica e guidata, ed è il sistema stesso che si assume tutto l'onere dei lavori di contorno, come l'individuazione delle parti di Documentazione e l'inserimento del documento nei nodi opportuni, demandando all'utente compiti ben più impegnativi a livello concettuale che riguardano il contenuto della Documentazione stessa. Anche in questa attività MIDA facilita l'utente permettendogli di venire a conoscenza in ogni momento della struttura precostruita di termini significativi che potrà usare, attraverso la loro selezione guidata durante la costruzione del documento.
D'altra parte il modo sistematico di procedere nella stesura del documento costringe l'utente, qualora il documento sia un programma, non solo a documentare il proprio lavoro (compito spesso dimenticato in quasi tutti glia ambienti) ma permette anche, attraverso la documentazione, di valutare il significato e la consistenza degli obiettivi posti per il programma che sta per realizzare. Per questo MIDA può essere considerato anche una metodologia per lo sviluppo di software.

Per quanto riguarda la portabilità possiamo affermare che, nonostante il riferimento fisico dei documenti rispetti le regole del file system di UNIX, la struttura dati in cui questi documenti sono inseriti è totalmente svincolata dal sistema operativo della macchina ospite. Oltre a ciò, anche la scelta del linguaggio PASCAL-UCSD contribuisce a rendere MIDA quasi totalmente trasportabile.

Attualmente MIDA è installato, a scopo sperimentale, su un sistema dedicato alle attività didattiche degli studenti del Corso di Laurea in Scienze dell'Informazione.

Si ringrazia il Prof. G. Degli Antoni per i preziosi suggerimenti e il Dr. P. Bonacina per il contributo tecnico durante la sua attività di tesi.

Questo lavoro è stato svolto nell'ambito del progetto "Metodologia di documentazione di programmi didattici" del M.P.I.

4. BIBLIOGRAFIA

[1] Leclerc, Zucker, Leclerc, A BROWSING APPROACH TO DOCUMENTATION, IEEE, 1982

[2] Croft, Pezarro, TEXT RETRIEVAL TECHNIQUES FOR THE AUTOMATED OFFICE, 1982

[3] Lockeed Information Systems, A BRIEF GUIDE TO DIALOG SEARCHING, Palo Alto, California, 1976

[4] Lucarella, AN END-USER ORIENTED TIME SHARING SYBSYSTEM FOR DATA BASE QUERY, Note di Software 17/18, 1982

[5] Gobbi, Maiocchi, AUTODOCUMENTAZIONE DEI PROGRAMMI, Note di Software, 3, 1977

[6] Le Van Huu, UN SEMPLICE APPROCCIO ALLA GESTIONE DI BIBLIOTECHE, 1977

[7] M. Ferrari, D. Marini, IL LABORATORIO DIDATTICO DI SCIENZE DELL'INFORMAZIONE: PRIME RIFLESSIONI E ALCUNI DATI STATISTICI, Convegno su "L'Università e l'evoluzione delle Tecnologie Informatiche"

[8] M. Maiocchi, R. Polillo, I GRUPPI DI PROGETTO NEI CORSI DI PROGRAMMAZIONE: ALCUNE ESPERIENZE, Convegno su "L'Università e l'evoluzione delle Tecnologie Informatiche

[9] P. Bonacina, STUDIO, PROGETTAZIONE E REALIZZAZIONE DI UN SISTEMA INTERATTIVO DI DOCUMENTAZIONE AUTOMATICA, Testi di Laurea in Scienze dell'Informazione, Università degli Studi di Milano.

Honeywell

Honeywell Information Systems Italia

Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA
Via Viotti, 5 - Milano